

Swift

Programe para iPhone e iPad



Casa do
Código

—  —
SÉRIE CAELUM

GUILHERME SILVEIRA
JOVIANE JARDIM

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

Sumário

1	Introdução	1
1.1	Motivação: mobile	1
1.2	Motivação: boas práticas e code smells	2
1.3	Agradecimentos	2
2	Projeto: nossa primeira App	5
2.1	Instalando o Xcode	5
2.2	Nossa primeira App	6
2.3	ViewController de cadastro de refeição	9
2.4	Boa prática: placeholder e keyboard type	23
2.5	Conectando a interface ao código	24
2.6	Conectando variáveis membro à sua parte visual: @IBOutlet	32
2.7	Resumo	38
3	Swift: a linguagem	39
3.1	Brincando no playground	39
3.2	Arrays	48
3.3	Boa prática: cuidado com inferência de tipos	52
3.4	Resumo	53
4	Swift: mapeando nossos modelos e introdução à Orientação a Ob- jetos	55
4.1	Boa prática: organizando os dados	55
4.2	Boa prática: Good citizen (evite nulos) e o init	64

4.3	Nosso Good Citizen	64
4.4	Code smell (mal cheiro de código): opcional!	70
4.5	Resumo	71
5	Projeto: organizando modelos com OO e arquivos em grupos	73
5.1	Boa prática: crie um diretório por grupo	74
5.2	Movendo arquivos no sistema operacional	78
5.3	Aplicando o básico de Orientação a Objetos	83
5.4	Resumo	87
6	Listando as refeições com TableViewController	89
6.1	Tabelas dinâmicas usando Dynamic Prototypes	97
6.2	Resumo	107
7	Projeto: lista de refeições	109
7.1	Resumo	113
8	Navegando entre telas	115
8.1	Navegando com uma pilha de telas	119
8.2	Resumo	121
9	Design pattern: delegate	123
9.1	Configurando um delegate via segue	127
9.2	Code smell: nomes genéricos demais e como extrair um protocolo	131
9.3	Resumo	135
10	Relacionamento um para muitos: lista de alimentos	137
10.1	Protocolos da API	141
10.2	Seleção múltipla e Cell Accessory	142
10.3	Desseleccionando elementos	146
10.4	Armazenando a seleção	146
10.5	Resumo	152

11	Criando novos itens	153
11.1	Pushando views para a pilha programaticamente	163
11.2	Voltando de um push programático	165
11.3	Invocando um delegate de forma programática	168
11.4	Resumo	178
12	Mostrando os detalhes de uma refeição com Long press	179
12.1	Recuperando onde ocorreu um evento de long press	182
12.2	Mostrando os detalhes em um alerta	186
12.3	Mostrando os detalhes dos itens	193
12.4	Resumo	196
13	Alerta: optionals e erros	197
13.1	Boa prática: evite optional, garanta tudo com if let	200
13.2	Tratando o erro com uma mensagem novamente	200
13.3	Boa prática: mensagens de erro descritivas	202
13.4	Refatoração: tratando diversos erros	204
13.5	Parâmetros default	206
13.6	Boa prática: Single Responsibility Principle, Princípio de Responsabilidade Única	208
13.7	Casos mais complexos de tratamento de erro	209
13.8	Code smell: Nested ifs	210
13.9	Resumo	214
14	Removendo uma refeição	215
14.1	Ações destrutivas e estilos	215
14.2	Passando uma função como callback	217
14.3	Identificando a linha a ser removida	220
14.4	Removendo e atualizando a tela	223
14.5	Closures	224
14.6	Code smell: closures a rodo	225
14.7	Resumo	229

15	Armazenando as refeições dentro do file system	231
15.1	Salvando as refeições no sistema de arquivos	237
15.2	Salvando e lendo itens no sistema de arquivos	242
16	Boa prática: dividindo responsabilidades e o Data Access Object	249
17	Aonde chegamos e próximos passos	255

Versão: 17.9.9

CAPÍTULO 1

Introdução

1.1 MOTIVAÇÃO: MOBILE

O mercado iOS cresce cada vez mais no Brasil, e se a decisão de uma empresa ou indivíduo é a de criar aplicações nativas, a escolha prática para o mundo iOS acaba sendo entre Objective C ou Swift.

Como uma linguagem nova e beta, Swift ainda possui espaço para pequenas mudanças que podem alterar a maneira de um programador desenvolver uma aplicação, mas nesse instante a Apple já a considera madura o suficiente para que novos aplicativos possam ser criados com ela.

Por ser nova, foram trazidos conceitos que estão em voga na comunidade de desenvolvimento em geral, como podemos ver a influência da linguagem Scala em Swift.

Como uma linguagem compilada e com uma IDE rica, recebemos muitas notificações de possíveis erros ainda em tempo de desenvolvimento, o que

evita diversos erros tradicionais, como o acesso a ponteiros inválidos na memória, algo muito fácil de se proteger em Swift.

Para desenvolvedores novos no mundo iOS, este livro busca ser um guia que ensina diversas partes da linguagem e da API disponível durante a criação e manutenção de uma aplicação mobile completa.

Todos os níveis de desenvolvedores podem se beneficiar dos conceitos de boas práticas, *code smells*, refatorações e *design patterns* apresentados no livro.

1.2 MOTIVAÇÃO: BOAS PRÁTICAS E CODE SMELLS

O objetivo desse livro não é somente guiá-lo através de sua primeira aplicação iOS, mas sim de ser capaz de julgar por si só o que é uma boa estratégia de programação, por meio da introdução de uma dezena de boas práticas e *code smells* que facilitam ou dificultam a manutenção do código com o passar do tempo.

Um code smell é um **sinal forte** de que existe algo de estranho no código, não uma garantia de que exista um problema. Da mesma forma, boas práticas e design patterns possuem situações específicas que podem trazer mais mal do que benefícios. Como fazemos no livro, toda situação encontrada deve ser analisada friamente: qual o custo de manter o código como está e qual o custo de refatorá-lo?

Saber pesar e tomar decisões como essa diferenciam um programador iniciante de um profissional e é isso que buscamos aqui: não somente ensinar a programar, mas sim criar profissionais em nossa área.

São dezenas de boas práticas, design patterns e code smells catalogados no decorrer do livro que podem ser acessados diretamente através do índice remissivo.

1.3 AGRADECIMENTOS

Gostaria de agradecer ao desafio proposto pelo Paulo Silveira e o Adriano Almeida. Não é fácil escrever um livro de boas práticas de linguagem e API quando uma é tão nova e a outra carregada de decisões antigas. É delicado entender as implicações de cada decisão da linguagem, mas o aprendizado que passamos por este projeto é o que trouxe a ele tanto valor.

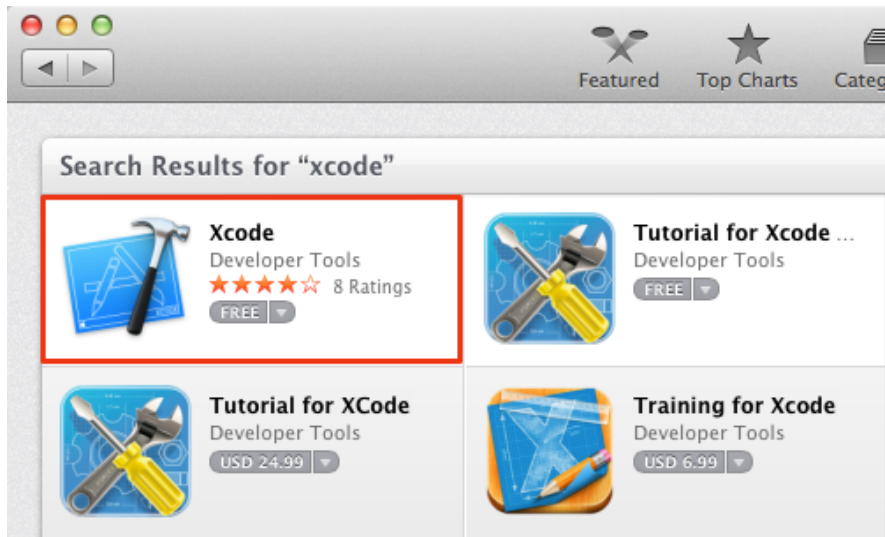
Um agradecimento especial ao Hugo Corbucci que tanto nos ajudou na revisão do livro, e ao Rodrigo Turini, ambos compartilharam conosco os bugs, as dificuldades e as alegrias de utilizar e ensinar uma linguagem ainda em desenvolvimento. Agradecemos também pelas conversas e discussões de padrões e boas práticas com o Maurício Aniche, além do Francisco Sokol, Diego Chohfi, Ubiratan Soares e outros.

CAPÍTULO 2

Projeto: nossa primeira App

2.1 INSTALANDO O XCODE

O processo de instalação do Xcode se tornou bem simples, basta acessar a Apple Store e procurar pelo Xcode. Clique em instalar.



Se você deseja usar uma versão beta do Xcode, entre no programa de *beta developers* da Apple e siga o link de download de uma versão beta. Cuidado, versões beta podem sofrer alterações e quebrar a qualquer instante - e quebram.

2.2 NOSSA PRIMEIRA APP

Nossa aplicação será um gerenciador de calorias e felicidade. Como usuário final eu faço um diário das comidas que ingeri, indicando quais alimentos estavam dentro dela e o quão feliz fiquei. Meu objetivo final seria descobrir quais alimentos me deixam mais felizes com o mínimo de calorias possíveis, um paraíso.

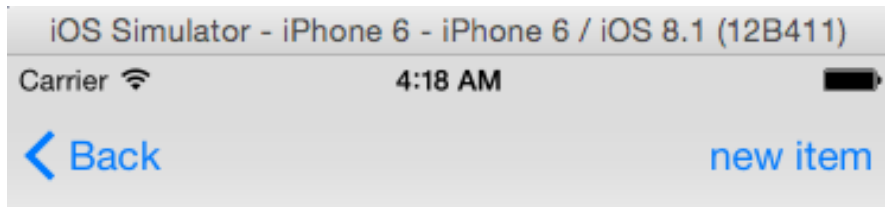
Para isso será necessário cadastrar refeições (`Meal`) e para cada refeição queremos incluir os itens (`Item`) que a compõem. Desejamos listar essas refeições e fazer o relacionamento entre esses dois modelos, afinal uma refeição possui diversos itens, além de armazenar todos esses dados no celular.

Por fim, veremos o processo de execução de nossa app em um simulador,

e o de deploy de nossa app tanto em um celular particular para testes quanto na app store.

Isso tudo será intercalado com seções de boas práticas de desenvolvimento de softwares e caixas de informações extras, ambos ganchos para que você possa se tornar um bom programador à medida que pesquisa a informação contida nessas seções e se aprofunda nelas. Não se sinta obrigado a pesquisá-las no mesmo momento que as lê, sugiro, inclusive, que você termine primeiro o conteúdo aqui apresentado para só então se aprofundar. Dessa forma, terá uma opinião mais formada sobre diversos itens ao entrar nessas discussões avançadas sobre qualidade de código, usabilidade etc.

Ao término de nossa jornada, teremos uma aplicação com diversas funcionalidades, entre elas, adicionar novas refeições:



Name

Happiness

[Add](#)

Eggplant Brownie

Zucchini Muffin

Cookie

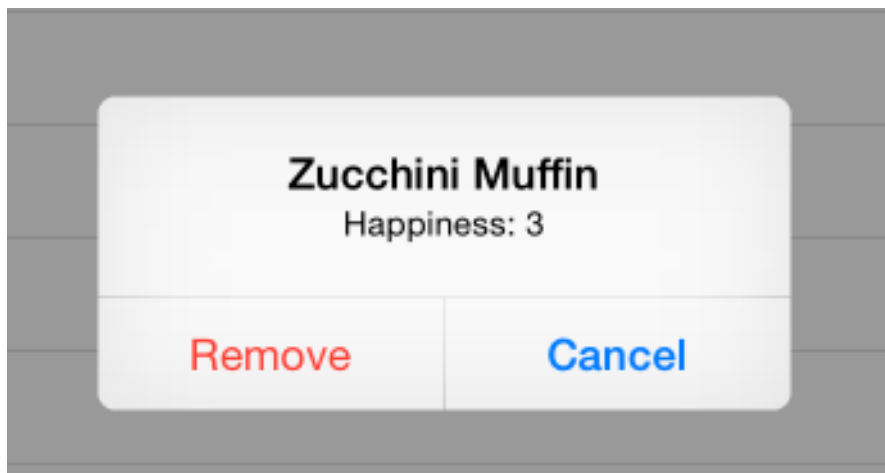
Coconut oil

Chocolate frosting

Chocolate chip

sundubu

Visualizar seus detalhes e removê-las desejado:

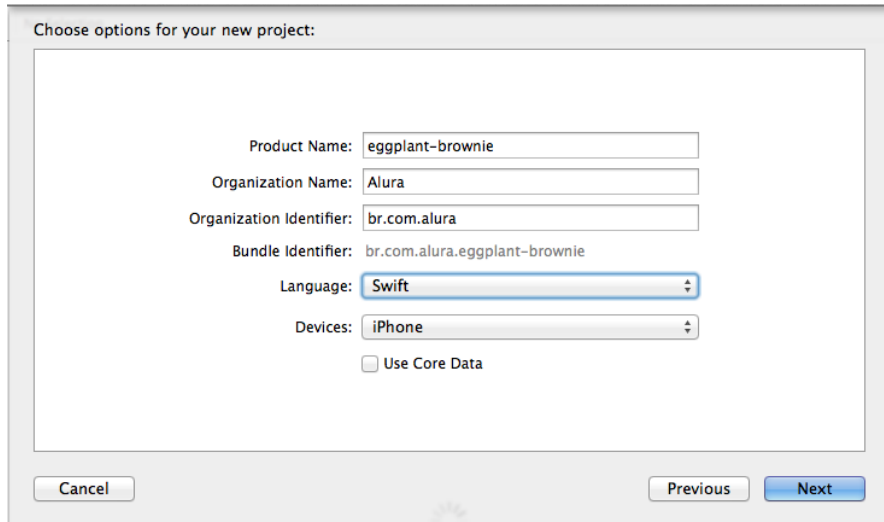


Pronto para começar?

2.3 VIEWCONTROLLER DE CADASTRO DE REFEIÇÃO

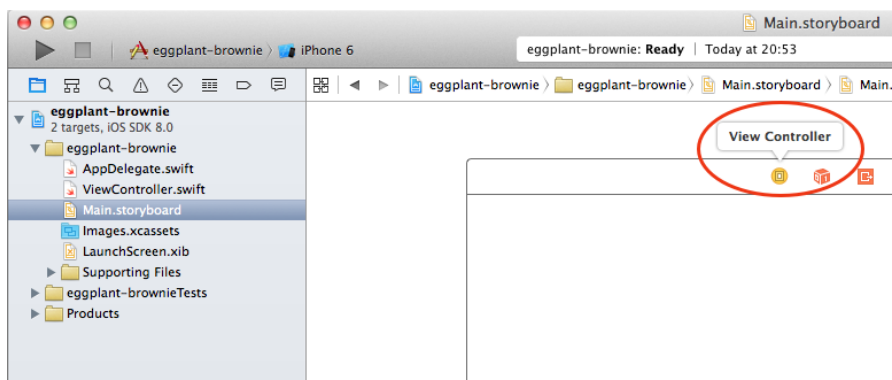
Vamos criar uma nova aplicação. No Xcode, escolhemos “Criar um novo projeto” e, no Wizard que segue, escolhemos `iOS, SingleViewApplication`, que seria uma aplicação de uma tela só. Aos poucos, evoluiremos a mesma para mais telas. O nome do nosso projeto é `eggplant-brownie` e a organização `br.com.alura`.

Escolhemos a linguagem `swift` e nosso alvo será o `iPhone`. A tela de criação fica então como a seguir:

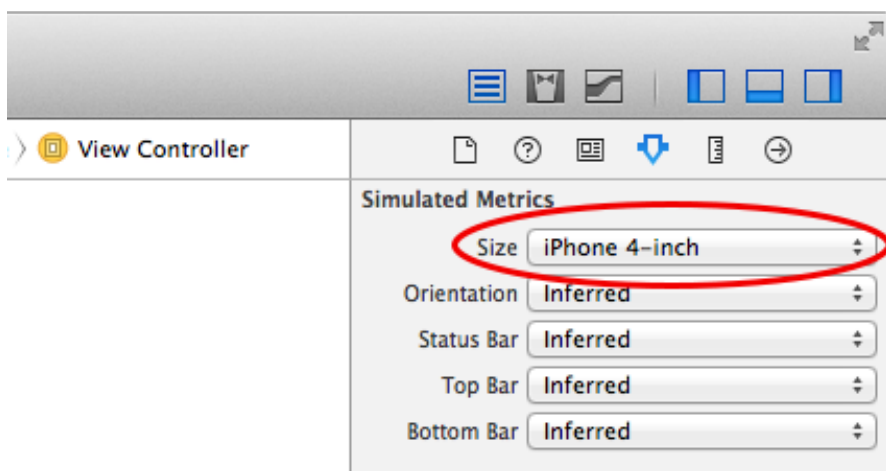


Logo de cara abriremos a nossa janela de criação de interface principal: o *storyboard* (chamado `Main.storyboard` dentro da pasta `eggplant-brownie`). Ao clicarmos nele, somos capazes de ver uma janela quadrada que será a primeira tela de nossa aplicação: nosso `ViewController`. O `View Controller` é chamado assim tanto por ser responsável pela `View` quanto por fazer o controle do que será executado ao interagirmos com essa `View`.

Tem algo de estranho aqui, nossa tela está quase quadrada e sabemos que um `iPhone` não é quadrado. Vamos escolher o *view controller* clicando no quadrado e, em seguida, no botão amarelo que aparece no topo à esquerda do quadrado:

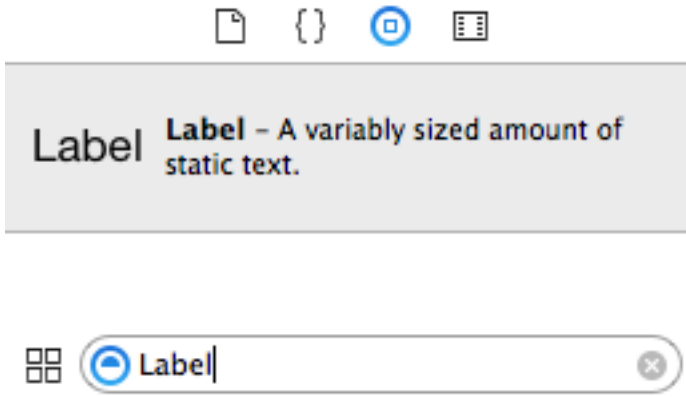


No lado direito, podemos navegar entre diversas propriedades do atual elemento escolhido. Na aba `Attributes`, temos uma seção chamada `Simulated Metrics` onde alteramos a opção `size`, escolhendo `iPhone 4-inch`. Agora sim temos um tamanho compatível com um `iPhone` de 4 polegadas.

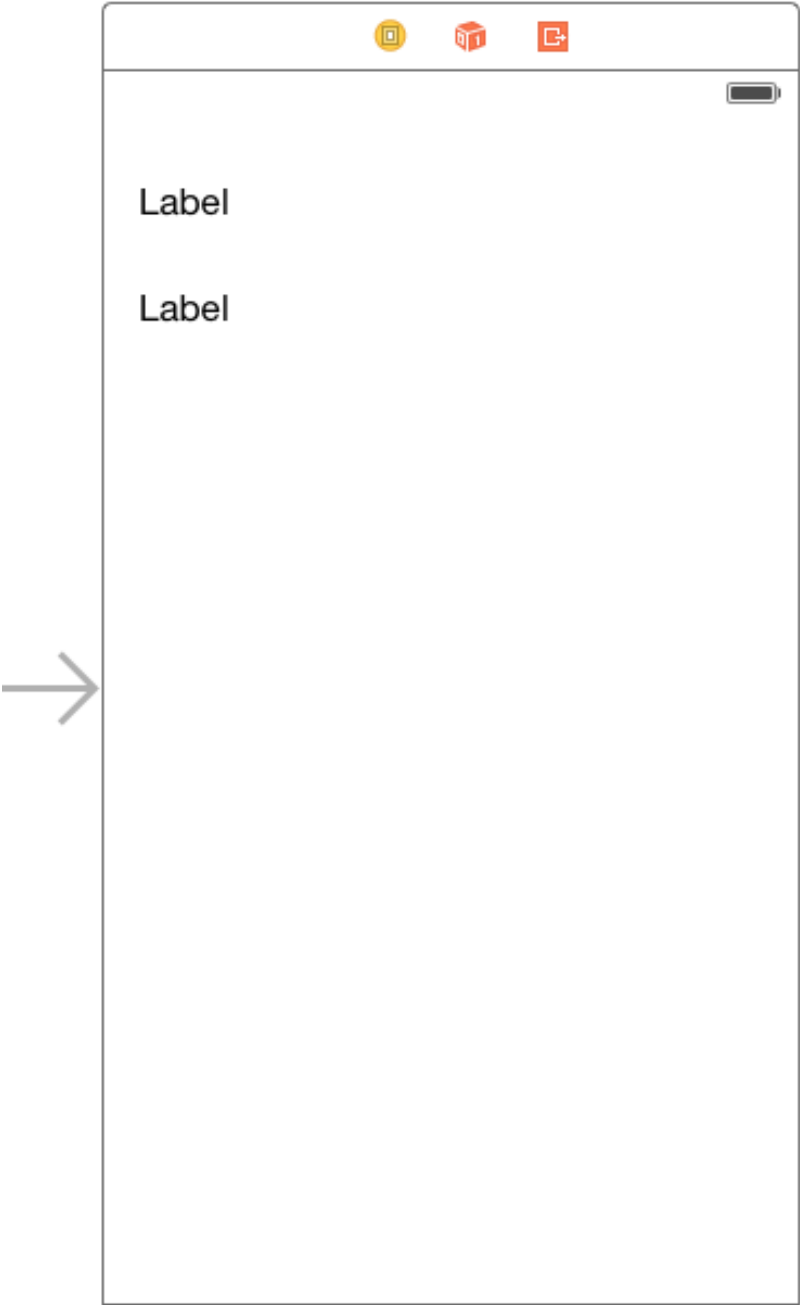


Desejamos adicionar duas mensagens de texto e dois campos de texto, um

para o nome da comida (`name`) e outro para o nível de felicidade com ela (`happiness`). Primeiro, buscamos na barra de componentes (canto inferior direito) um campo chamado `Label`:



E agora o arrastamos duas vezes para nosso `ViewController`. O resultado são dois `labels` com nomes sem sentido:



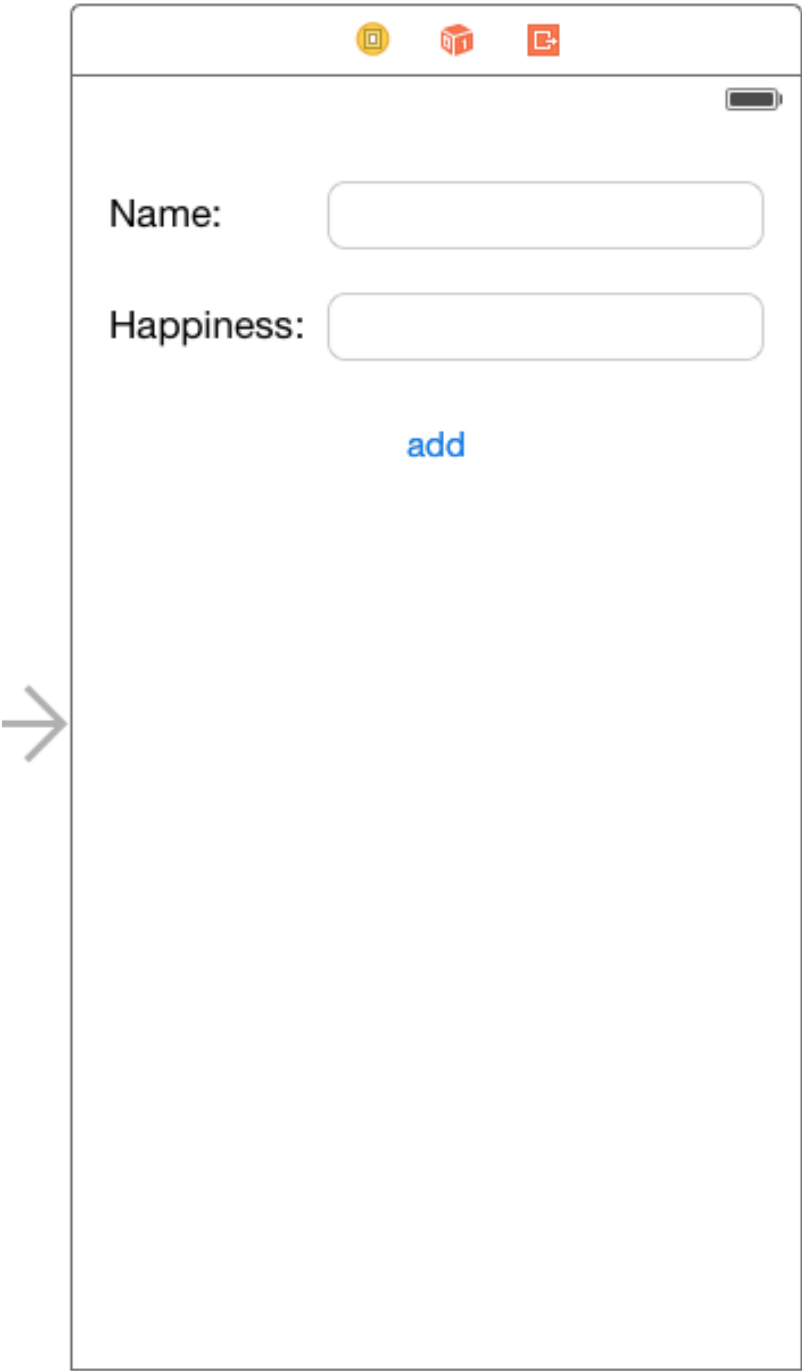
Precisamos trocar seus valores. Afinal, um representará o nome e o outro nosso nível de felicidade com aquela comida. Um duplo clique no `label` permite alterar seu valor. Mudemos para `Name` e `Happiness`:



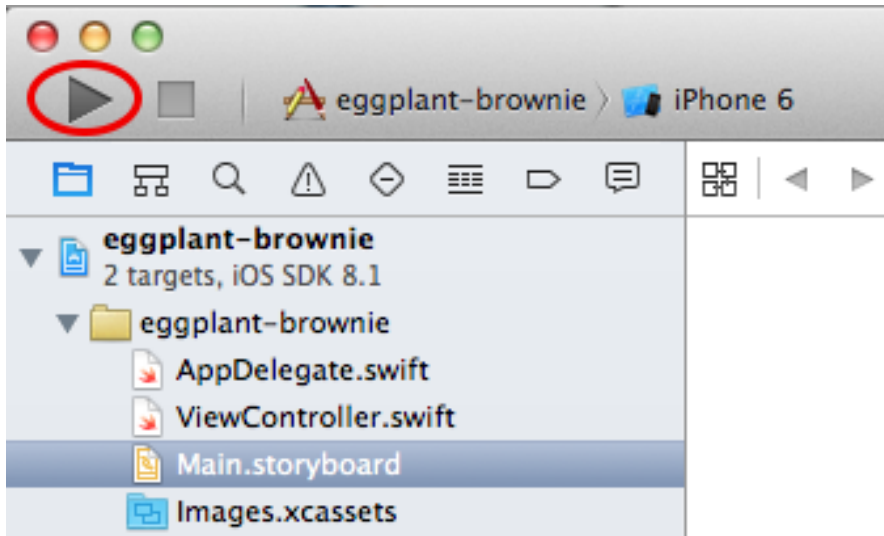
Adicionamos agora dois campos de `Text Field` da mesma forma, um para `Name` e um para `Happiness`. No mesmo canto inferior direito, procuramos por `Text Field` e arrastamos o resultado duas vezes.

Colocamos um botão, procurando pelo componente chamado `Button` e, assim como com outros componentes, mudamos seu texto para `add`.

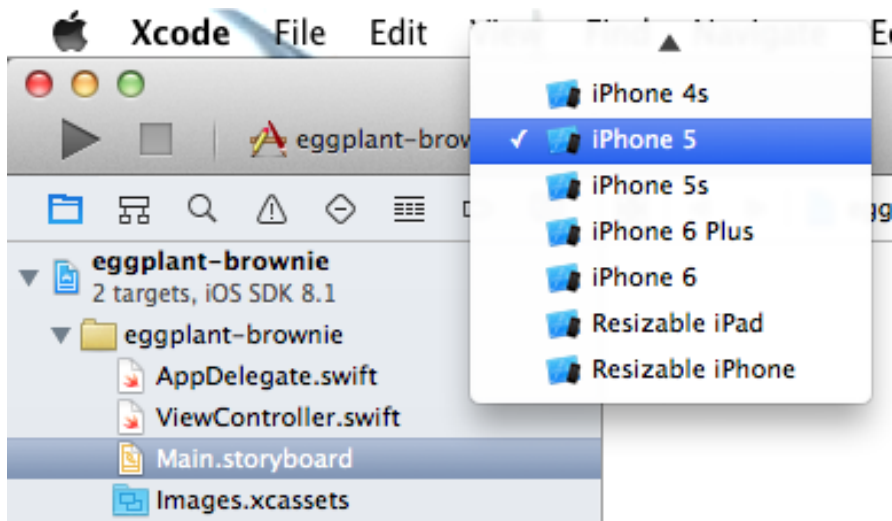
O resultado de nosso `storyboard` é esse:



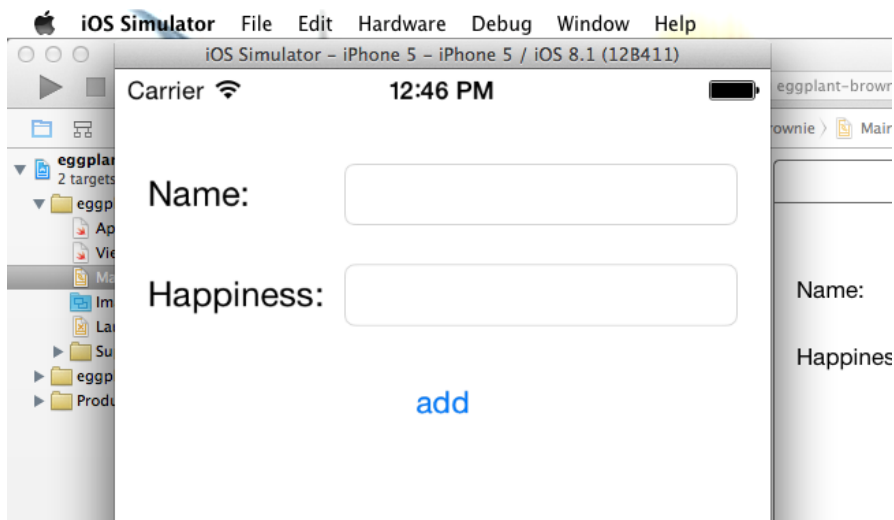
O próximo passo é rodar nosso programa em um iPhone. Mas calma lá, eu não tenho um iPhone 4, 5, 6 etc., a todo instante. Quero primeiro ver como ficaria em um iPhone no meu computador, então vamos simular um iPhone. Para isso clicamos no botão `Play` que fica no canto superior esquerdo da janela.



Poderíamos rodar em outro iPhone, repare que podemos trocar o modelo a qualquer instante que desejarmos. Por exemplo, ao escolher um iPhone 5:



O resultado é a tela de nosso programa rodando:

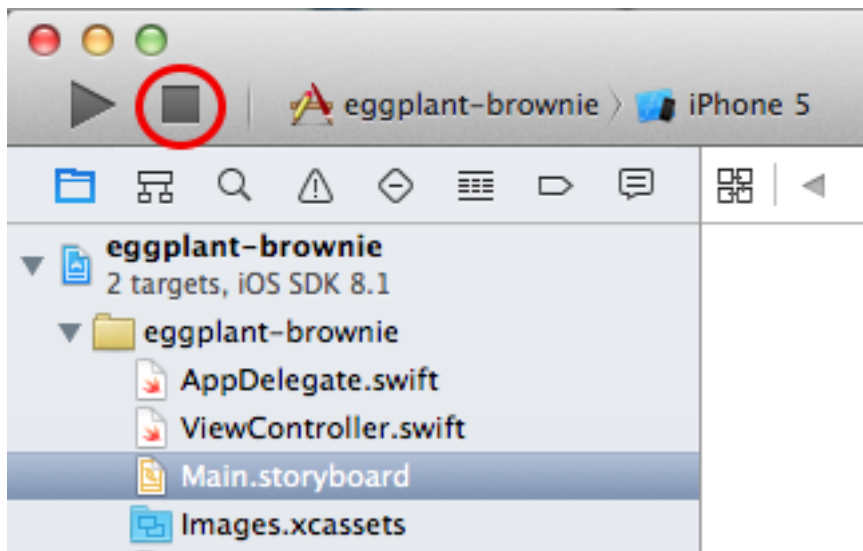


Podemos diminuir o zoom do emulador escolhendo, na aplicação `iOS Simulator`, menu `Window`, submenu `Scale` e a opção que acharmos mais adequada. Podemos também optar por mostrar o teclado em nosso emulador no menu `Hardware`, submenu `Keyboard` e `Toggle software keyboard`.

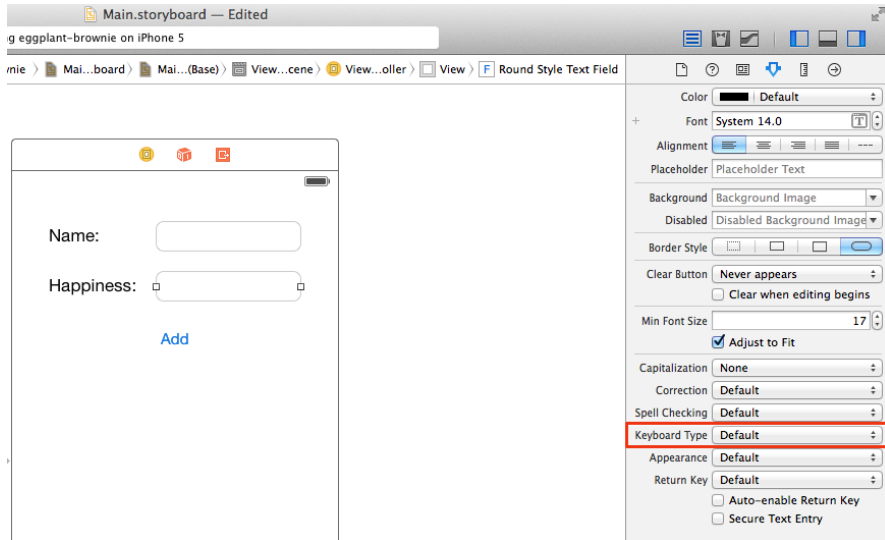
EMULADOR OU SIMULADOR?

Um emulador emula também o hardware, o simulador utiliza o hardware da máquina onde está rodando (host). No nosso caso, estamos usando um simulador, com o hardware de nossa máquina, portanto nossa aplicação rodará em geral mais rápido do que em um iPhone de verdade. Lembre-se sempre disso antes de colocar uma aplicação que abusa de processamento disponível para o mundo: você quer rodá-la em um emulador ou em seu celular antes para conferir seu desempenho.

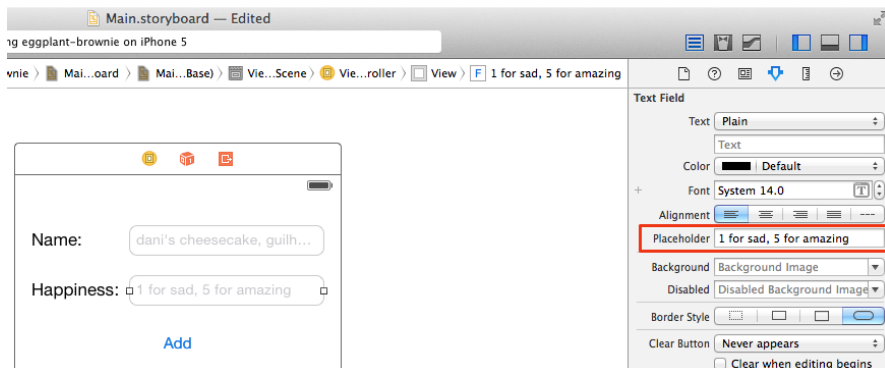
Ainda tem algo de estranho em nossa aplicação, o campo de felicidade permite digitar texto. Não desejamos isso. Podemos parar o emulador clicando no botão de `stop` no canto superior esquerdo.



Ao selecionarmos um campo de texto como o de felicidade, notamos que à direita, na aba de propriedades, ao final das informações para `Text Field`, temos uma opção que indica o tipo de teclado, o `Keyboard Type`:



Vamos escolher o tipo chamado `Number Pad`. Escolhemos também um texto padrão para ficar no fundo dos dois campos de texto, um `placeholder`, que será `dani's cheesecake, guilherme's sundubu etc` e `1 for sad, 5 for amazing`.



Rodamos nosso programa clicando novamente no `Play` e, agora, ao clicarmos no campo de texto do número, podemos verificar que o teclado que aparece é o teclado numérico, facilitando bastante a entrada de informações pelo usuário final.

2.4 BOA PRÁTICA: PLACEHOLDER E KEYBOARD TYPE

Formulários cuja entrada de dados não seja do dia a dia do usuário final podem deixá-lo perdido. Um formulário de login é tão comum que não precisa de `placeholder` e a tela fica mais limpa. Já um `placeholder` em um formulário complexo que o usuário vê pela primeira vez pode ajudá-lo a entender que tipo de valor o programa espera dele.

Além do `placeholder`, o tipo de teclado ajuda o usuário ao digitar dados específicos como e-mails, números inteiros ou decimais.

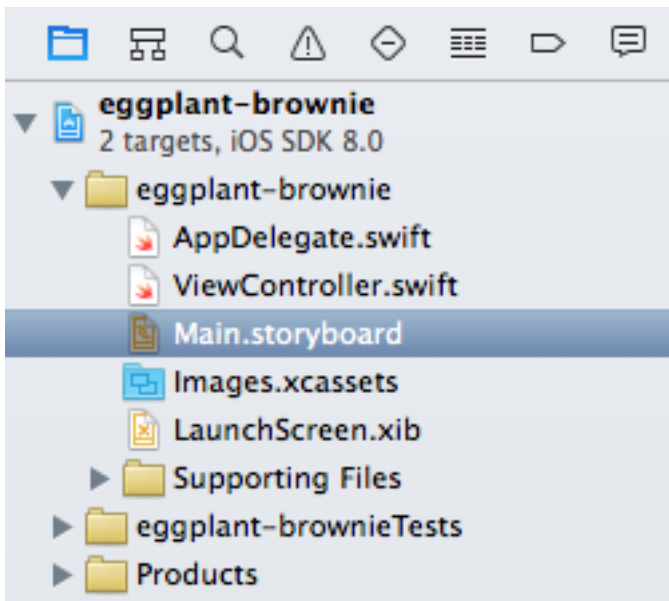
Sempre que adicionar um campo de texto em sua aplicação, lembre-se de configurar o `placeholder` e o `keyboard type` adequado.

DICA: CUTUQUE AS PROPRIEDADES

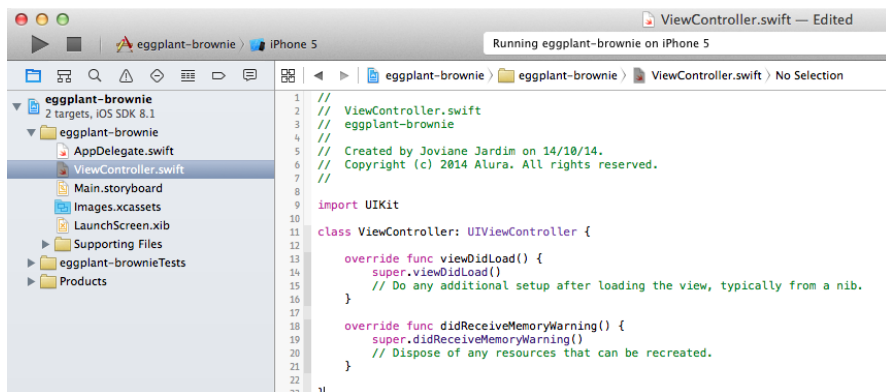
Brinque com algumas propriedades de fonte de seu componente `Label`. Mas lembre-se: é raro que uma aplicação com muitos estilos de fonte seja lembrada por esse fator. Evite muitas variações de fonte em sua aplicação. Não crie 100 estilos diferentes para 3 telas.

DICA: SALVANDO SEUS ARQUIVOS

Note que toda vez que rodamos nosso programe ele é salvo automaticamente. Para perceber se seu arquivo não está salvo, você verifica que o nome dele está em negrito ou seu ícone está escurecido:

**2.5 CONECTANDO A INTERFACE AO CÓDIGO**

Mas o que acontece quando clicamos no botão `add`? Desejamos executar algum código ao clicar neste botão. Para isso, abrimos o arquivo `ViewController.swift`, onde definiremos uma função em Swift. Clicamos no nome do arquivo na barra esquerda:



Quem é ele? Prazer, nosso `ViewController`, que já vem um comportamento mínimo:

```
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view,
        // typically from a nib.
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }
}
```

Os dois métodos que a IDE adicionou são opcionais, portanto os removemos para ficarmos com nosso `ViewController` puro:

```
import UIKit

class ViewController: UIViewController {
```

```
}
```

O básico da linguagem aqui é que um `ViewController` herda de `UIViewController`. Só isso! O que queremos é adicionar uma função que será invocada quando o botão `add` seja clicado. Nada mais natural que definirmos nossa função como `add`. Toda vez que o botão for clicado, a função poderá ser chamada.

```
import UIKit

class ViewController: UIViewController {
    func add() {

    }
}
```

Queremos que essa função seja invocada toda vez que acontecer algo com nossa interface com o usuário. Isto é, gostaríamos que essa função fosse um *listener* dos eventos da nossa *view*. Portanto, nós o anotamos de modo a dizer ao Interface Builder (IB) que este método de nossa classe de `ViewController` é uma ação (*Action*) que será informada pela tela. O método é anotado como `@IBAction`:

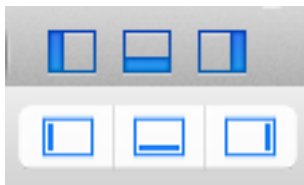
```
@IBAction func add() {

}
```

Para testar nosso *callback*, adicionamos uma impressão ao log:

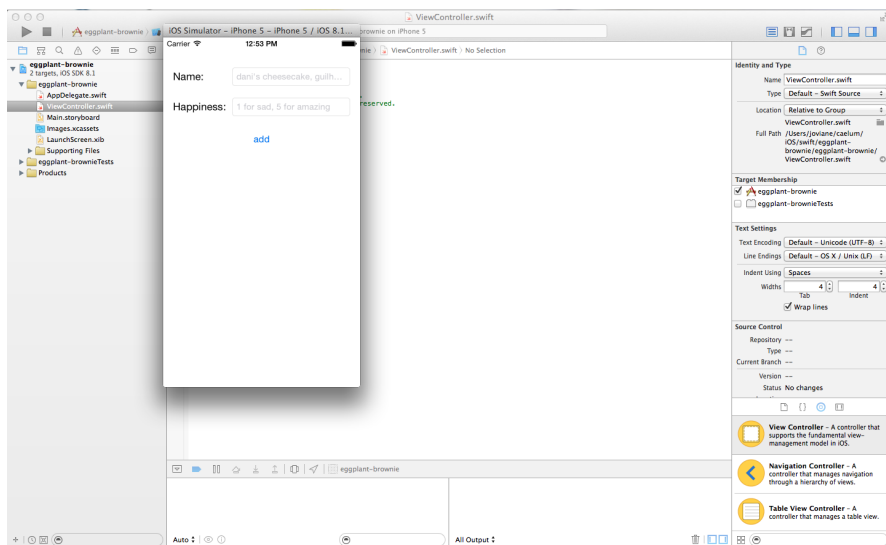
```
@IBAction func add() {
    println("button pressed!")
}
```

O log ficará na parte de baixo da tela, e aparecerá automaticamente ao imprimirmos algo, ou podemos mostrá-lo acessando o menu `View`, submenu `Debug Area`, opção `Hide/Show Debug Area`.



Esses 3 botões (os de baixo para Yosemite e mais recente, os de cima para mais antigos) que se encontram no canto superior direito da janela do Xcode servem para esconder diferentes barras da aplicação: a barra de navegação nos arquivos do projeto (`Navigator`, `Command+1` para mostrar, `Command+0` para esconder), a barra de propriedades (`Utilities`) e a barra que mostra o console e o debug embaixo (`Debug`). Utilize a cada instante o modo que for mais agradável para a tarefa que está efetuando.

Rodamos o programa, como sempre, clicando em `Play`, e, na aplicação, clicamos no botão `add`, mas nosso log na IDE não mostra nada:

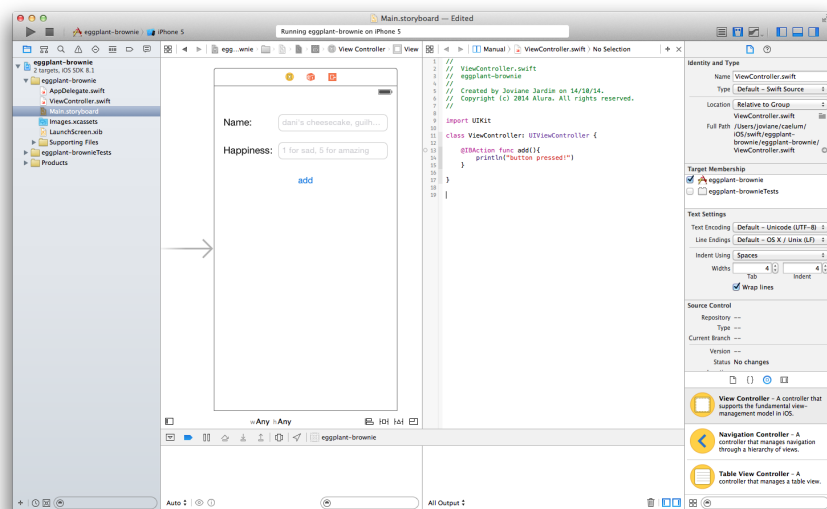


Claro, nós ainda não conectamos o botão `add` com nossa função. Podemos fazer isso de diversas maneiras e a primeira que veremos aqui é feita arrastando o código para o botão visualmente. Mas como arrastar algo visual para o código se a IDE mostra somente uma coisa por vez?

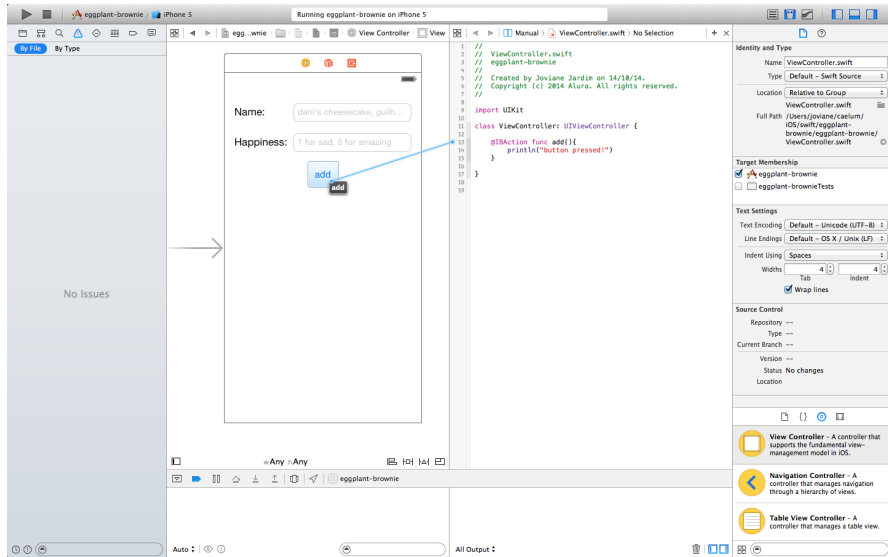
No topo, à direita do Xcode, ao lado dos ícones para exibir/esconder as barras, conseguimos modificar a maneira de visualizar nossa área de trabalho. O primeiro botão ativa a visualização tradicional que utilizamos até agora. O segundo ativa a visualização que permite abrir dois editores ao mesmo tempo e é nela que costumamos trabalhar enquanto desenvolvemos a parte visual de nossas apps. Vamos clicar nesse botão.



Abrimos o storyboard clicando na barra à esquerda e clicamos no nosso `ViewController` dentro dele. Automaticamente o assistente (a parte da direita) abre o código-fonte do controller:



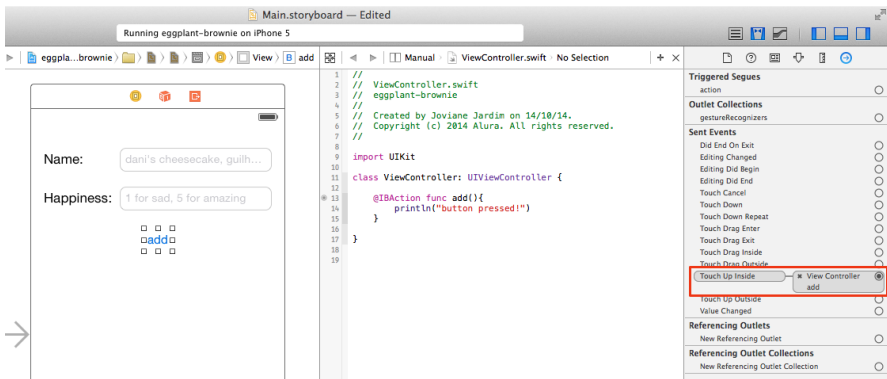
Note a bola vazia ao lado de nosso `@IBAction`. É ela que arrastamos para conectar ao botão `add`:



Testamos nossa aplicação e, ao clicarmos no botão, o log indica que o evento ocorreu. Sucesso:



De volta ao Xcode, ao selecionarmos o botão `add`, podemos verificar na última aba de propriedades (Connections Inspector - último ícone à direita) que a ação `Touch Up Inside` invocará nossa função.



Poderíamos acessar esta listagem de eventos (*Sent Events*) e puxar qualquer um dos eventos para uma função de *callback* que definimos. No entanto, não necessitamos de mais nenhuma agora.

N MANEIRAS: O MESMO RESULTADO

Essas são só duas das maneiras de trabalhar com o Xcode para criar uma `@IBAction`, e existem outras que veremos. Mesmo que no final todas tenham o mesmo resultado, tentaremos mostrar diversas delas para que você possa escolher aquela em que sentir mais prática de acordo com o que você tem em mãos aberto em sua tela naquele instante - ou de quais configurações deseja fazer ao criar uma `IBAction` ou similares.

Chegou a hora de ler os dados dos dois campos, *Name* e *Happiness*. Dentro de nossa função, vamos imprimir os dois, primeiro declarando duas *Strings* e concatenando-as durante a impressão:

```
@IBAction func add() {
    var name : String = "guilherme's sundubu";
    var happiness : String = "5";
    println("eaten: \(name) \(happiness)!");
}
```

Note que em Swift o `;` é opcional, ele só é obrigatório quando desejar utilizar duas expressões na mesma linha, portanto podemos remover os `;`:

```
@IBAction func add() {  
    var name : String = "guilherme's sundubu"  
    var happiness : String = "5"  
    println("eaten: \(name) \(happiness)!")  
}
```

Como já estamos inicializando as variáveis no mesmo instante de suas declarações, não precisamos declarar seus tipos, elas têm seus tipos inferidos direto na inicialização:

```
@IBAction func add() {  
    var name = "guilherme's sundubu"  
    var happiness = "5"  
    println("eaten: \(name) \(happiness)!")  
}
```

Além disso, tanto o nome quanto a felicidade não mudam de valor, de modo que podemos declará-las como referências imutáveis, ou seja, constantes. Podemos fazer isso utilizando a palavra reservada `let`:

```
@IBAction func add() {  
    let name = "guilherme's sundubu"  
    let happiness = "5"  
    println("eaten: \(name) \(happiness)!")  
}
```

2.6 CONECTANDO VARIÁVEIS MEMBRO À SUA PARTE VISUAL: @IBOutlet

Voltando ao código, não queremos valores arbitrários como “guilherme’s sundubu” ou “5”. Desejamos ler o valor de nossos campos de texto. Como fazer isso? Da mesma maneira como conectamos uma função a um elemento visual, podemos conectar um elemento visual por inteiro a uma variável. No nosso caso, queremos representar um `UITextField`, portanto nossa variável será deste tipo:

```
var nameField: UITextField
var happinessField: UITextField
```

Cuidado, pois não declaramos essas variáveis como locais à função! Assim como a função, elas são membros de nossa classe, membros de nosso ViewController:

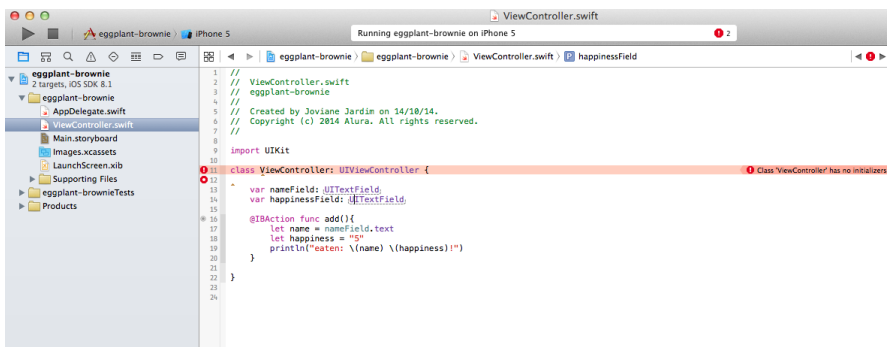
```
import UIKit

class ViewController: UIViewController {

    var nameField: UITextField
    var happinessField: UITextField

    @IBAction func add() {
        let name = "guilherme's sundubu"
        let happiness = "5"
        println("eaten: \(name) \(happiness)!")
    }
}
```

Mas, na hora em que a adicionamos, a IDE indica erros. Ao clicarmos no ícone vermelho que representa um erro na barra à esquerda, vemos a mensagem de que a Class ViewController has no initializers:



Isso ocorre pois toda variável membro deve ser inicializada ou declarada como opcional.

ERROS DE COMPILAÇÃO

Infelizmente o parser e compilador da linguagem Swift pode ser perder em seu código com uma certa facilidade. Caso o Xcode não mostre o erro mencionado, tente rodar sua aplicação, nesse instante é certo que ele tentará compilar seu código e encontrará o erro. Sempre que era para ter um erro de compilação e o Xcode não mostrar para você, tente rodar sua aplicação, forçando a compilação.

Colocamos o `!` ao final da declaração para indicar que ela é opcional para a criação de um `ViewController`.

```
var nameField: UITextField!  
var happinessField: UITextField!
```

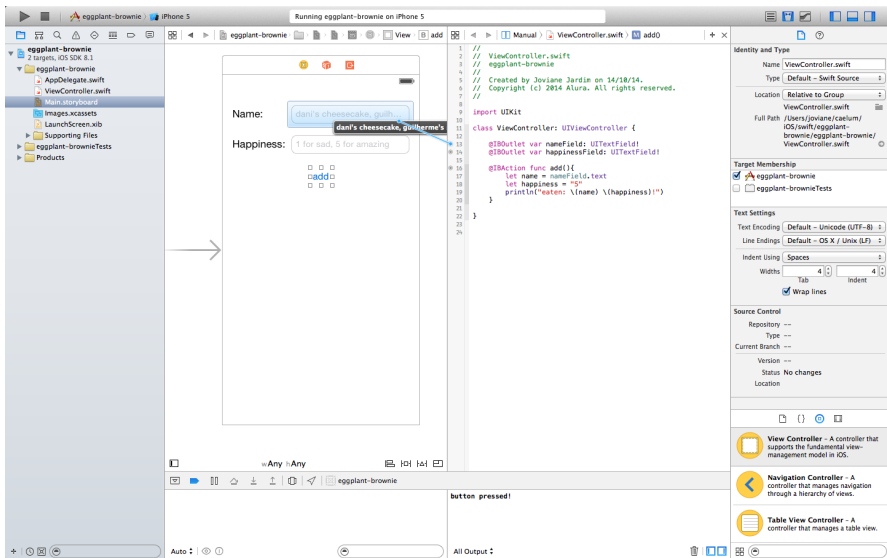
CUIDADO: OPCIONAIS COM !

Cuidado: uma variável declarada opcional com `!` que tenha valor nulo e seja acessada pode crashear sua aplicação. Veremos outra maneira de declarar e acessar variáveis opcionais, além de vantagens e desvantagens de cada abordagem no decorrer do conteúdo aqui apresentado.

Não desejamos programar de qualquer jeito e uma das principais boas práticas que veremos diversas vezes estará ligada ao uso adequado de valores opcionais para evitar erros em nossa aplicação. Não pretendemos ensinar de qualquer maneira a utilização de `!` nem mesmo incentivar o uso indiscriminado do mesmo: seremos bem críticos em relação a ele. Por enquanto, adotamos o `!` na declaração de nossas variáveis para indicar ao compilador que a variável é opcional mas nós, desenvolvedores, sabemos que ela tem valor (e corremos o risco).

Da mesma maneira como fizemos com nossa ação, devemos anotar nossas variáveis. Desta vez, nós as anotamos como `@IBOutlet` e puxamos a

bola vazia ao lado da variável `name` para nosso campo `name` na interface visual:



Agora, dentro de nossa função de adicionar uma nova refeição, podemos pegar o valor de nossos campos. Em cada um deles, utilizamos a propriedade `text` para extrair seus valores:

```
import UIKit
```

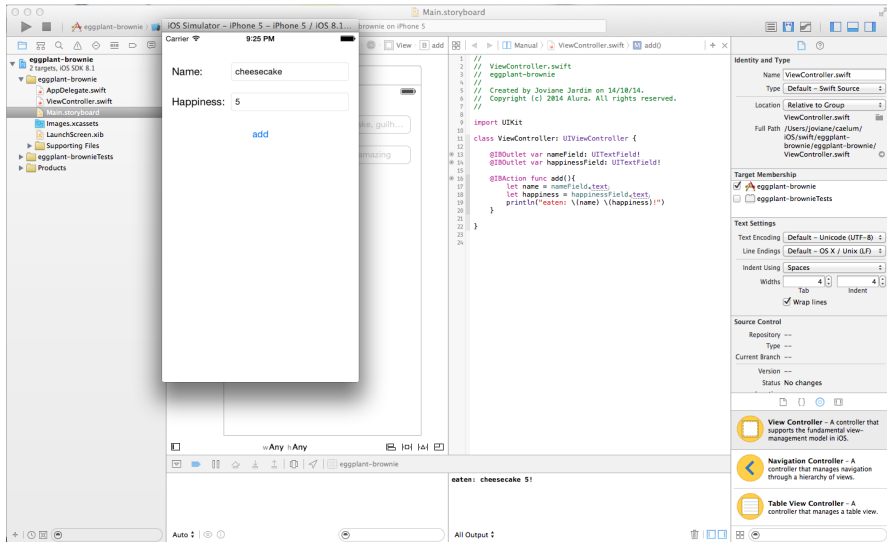
```
class ViewController: UIViewController {

    @IBOutlet var nameField: UITextField!
    @IBOutlet var happinessField: UITextField!

    @IBAction func add() {
        let name = nameField.text
        let happiness = happinessField.text
        println("eaten: \(name) \(happiness)!")
    }
}
```

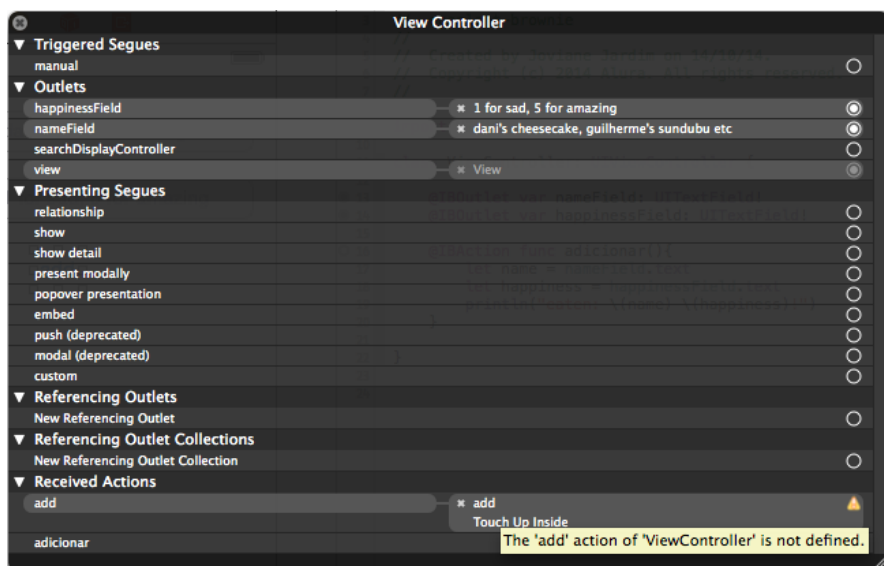
```
}
```

Ao clicarmos no botão, vemos o resultado no log com os valores que entramos nos campos. É o que esperávamos!



AÇÕES CONECTADAS

Preste sempre atenção ao desenho dos círculos (as bolinhas) ao lado de suas ações. Se um círculo estiver preenchido, ele está conectado a algo. Se ele estiver vazio, então está desconectado. Se você renomear o método, ele se desconectará pois a conexão é feita via o nome do método. Tome cuidado sempre que renomear métodos ou atributos anotados. Tenha cuidado também em copiar/colar componentes conectados pois a conexão também é copiada. Caso seja necessário renomear ou copiar/colar um item já conectado, lembre-se de desfazer a conexão efetuada anteriormente, pois o Xcode acusará um erro:



Se você rodar a aplicação com uma conexão mal feita, ela pode desde (sem querer) funcionar de maneira inesperada como parar.

NOME DOS CAMPOS

Existe um padrão comum em desenvolvimento de interfaces de programas desktop baseado na notação húngara, onde tentamos indicar no nome da variável qual o seu tipo.

Um combo box para representar o gênero de um usuário poderia se chamar `cbGender`, enquanto o campo de texto com seu nome seria o `txtName`. Esse padrão não é seguido como regra geral no desenvolvimento de aplicações iOS. Outro padrão comum é utilizar `field` como prefixo ou sufixo. Aqui adotamos `field` como sufixo, assim como é feito no nome das classes (`Controller`, `Delegate` etc) nas APIs do iOS.

2.7 RESUMO

Somos capazes de criar uma interface mínima com nosso usuário, além de receber valores e ter métodos invocados quando determinados eventos ocorrem.

Aprendemos um pouco da linguagem Swift: sabemos criar uma classe, variáveis locais e membros, além da definição de constantes. Vimos também que a linguagem é `type safe` inclusive em relação à não inicialização de variáveis: fomos obrigados a utilizar o `!` para dizer que sabemos que essa variável é opcional e terá um valor válido em execução.

Aprendemos também a usar nossa IDE para navegar no projeto e construir nossa interface, que foi rodada em um simulador do iPhone. Somos capazes de testá-la em diversas versões de iPhones.

Durante esse caminho todo conseguimos perceber algumas boas práticas de desenvolvimento de software em geral, tanto em relação à utilização de constantes quando cabível quanto em relação à interface que oferecemos para nosso usuário final.

Nosso próximo passo é aprender mais dessa linguagem para podermos modelar nossas classes que representarão uma refeição (`Meal`) e um item contido nela (`Item`).

CAPÍTULO 3

Swift: a linguagem

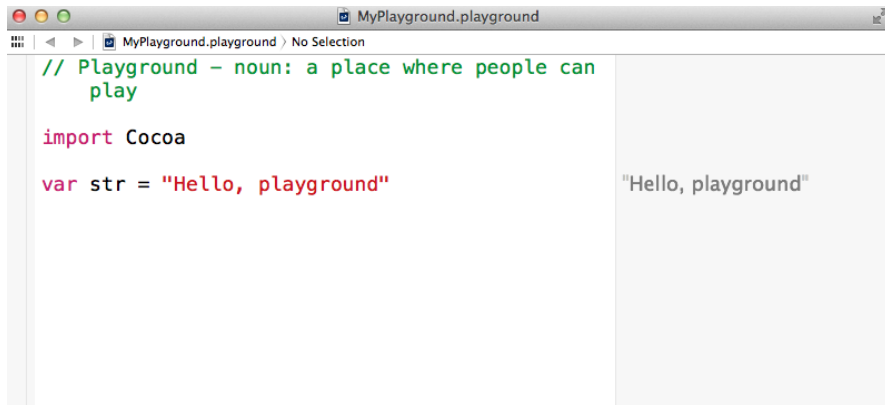
Veremos aqui a base da linguagem que será suficiente para criarmos uma aplicação utilizando diversas boas práticas de programação. Não nos viciaremos com o uso desnecessário de variáveis globais e outras más práticas que dificultam a manutenção do código ao longo prazo. Para isso utilizaremos diversos padrões, além do básico de Orientação a Objetos.

Nosso foco não é ensinar Orientação a Objetos (OO) avançada aqui, mas quanto maior sua base em boas práticas de OO de outras linguagens, melhor poderá tomar cuidados importantes em Swift. Diversos deles serão citados por aqui.

3.1 BRINCANDO NO PLAYGROUND

Criamos primeiro um playground, onde poderemos testar diversas funcionalidades. Na janela do Xcode, `Command+N` e escolhemos `iOS, playground`,

deixamos o nome `MyPlayground.playground` e temos o playground pronto! Na esquerda, escrevemos o código, na direita vemos o seu resultado:

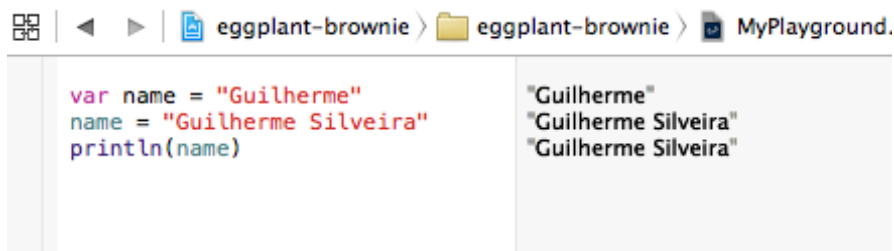


Note que o playground não faz parte do seu projeto em si, somente deve ser usado para testes rápidos, que é o nosso caso. Vamos utilizá-lo para entendermos melhor diversos conceitos da linguagem.

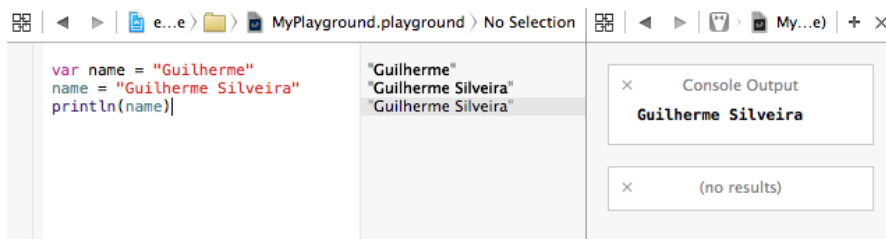
Primeiro, testaremos algumas coisas que já conhecemos. Removemos o código do playground atual e criamos uma `String` imprimindo-a:

```
var name = "Guilherme"  
name = "Guilherme Silveira"  
println(name)
```

Na direita, temos os resultados de nossas execuções:



Da mesma forma como usamos um editor com assistência para arrastar nossos `IBOutlet`s e `IBAction`s, o playground oferece um assistente chamado `Timeline`. Ele pode ser ativado clicando naquele ícone que usamos para abrir o editor para arrastar as ações ou no menu `View`, submenu `Assistant Editor`, opção `Show Assistant Editor`. Na saída dele, vemos o resultado de nossas informações de log:

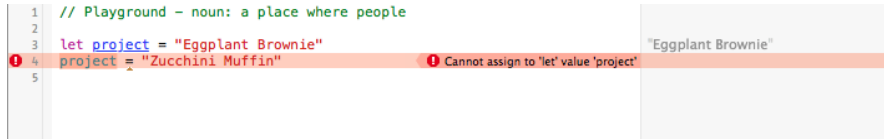


Como vimos antes, em Swift, o `;` no fim de uma instrução é opcional, ele só é obrigatório quando duas instruções ficam na mesma linha:

```
var name = "Guilherme"
name = "Guilherme Silveira"; println(name)
```

Podemos também criar uma constante. Se tentarmos atribuir um novo valor à constante, o compilador mostra uma mensagem de erro:

```
let project = "Eggplant Brownie"
project = "Zucchini Muffin"
```



BOA PRÁTICA: CONSTANTES

Utilizar constantes (em Swift, `let`), quando cabível, é considerado uma boa prática em programação em geral, ao invés de referências mutáveis (em Swift, `var`). Isso pois, uma vez definida, a variável continua com o mesmo valor até “deixar de existir” (oficialmente, ser desreferenciada). Não existe nenhuma complexidade ligada à troca de referências.

Como vimos, uma constante define uma conexão imutável entre seu nome e o valor passado durante sua inicialização. Não há preocupação de ela ter sido alterada antes de acessá-la em qualquer ponto de seu programa.

E se desejamos comentar alguma linha no nosso código? O comentário tradicional é o `//` que comenta o resto da linha:

```
// too complex code need a one line comment
let project = "too complex, so it requires a comment"
```

Outra opção seria usar o `/* ... */`, que permite o comentário em diversas linhas.

```
/*
nuclear devices code might need
multiple line comments
*/
let device = "nuclear"
```

O `/* ... */` pode ser usado também para comentar algo no meio da linha:

```
let universe /* needs a middle comment to explain */ = "big"
```

CODE SMELL: COMENTÁRIOS

Em geral, quando precisamos comentar algo, é sinal de que o código está complexo e o próximo desenvolvedor sofrerá para entendê-lo. Se o desenvolvedor sofre, a chance de surgirem novos erros por uma mudança indevida é grande. Isto é, a necessidade do comentário está indicando que o código que escrevemos está declarado de determinada maneira que não parece ser fácil de manter.

Sendo assim, esse `code smell` pode ser usado como um sinal para alterarmos nosso código. Temos como arsenal de refatoração diversas ações, como extrair variáveis, métodos, renomear ambos etc. Deixar nosso código claro o suficiente para o próximo desenvolvedor pode diminuir a chance de futuros bugs serem introduzidos em nossa aplicação.

Vamos aplicar a declaração de variáveis ao nosso domínio, de refeições e itens. Podemos apagar todo o nosso playground.

Primeiro, precisaremos representar uma refeição. Em inglês, uma refeição será representada por um `meal`, e ela tem tanto o nome que demos para a refeição daquele dia (`String`), quanto nossa nota, que será representada com um número inteiro (`Int`):

```
let name = "Dani's paradise"  
let happiness = 5 // Int
```

Representaremos uma refeição por um conjunto de alimentos, como uma sopa de abóbora ou seu brownie de berinjela, com poucas calorias. Para representar as calorias, usaremos um número decimal, um número com ponto flutuante:

```
let name = "eggplant brownie"  
let calories = 50.5 // Double
```

Mas repare que o código não compilará se você mantiver o exemplo anterior: duas variáveis não podem ter o mesmo nome (`name`) no mesmo escopo. Neste capítulo, antes de cada exemplo, lembre-se de apagar ou comentar o código anterior antes de continuar.

Além desses tipos, será recorrente o uso de variáveis do tipo `true` ou `false`, os tipos booleanos:

```
let eggplantBrownieIsVeggie = true
let eggplantBrownieTastesAsChocolate = false
```

Variáveis booleanas em Swift são do tipo `Bool`, portanto uma declaração com tipagem explícita seria:

```
let eggplantBrownieIsVeggie:Bool = true
let eggplantBrownieTastesAsChocolate:Bool = false
```

TIPOS BÁSICOS

Existem, claro, diversos outros tipos para utilizarmos, mas o básico que cobre a maior parte dos casos são os apresentados até aqui: `Double`, `Bool`, `Int` e `String`. Além desses, já vimos e continuaremos vendo outras classes que serão utilizadas em nosso projeto.

Há desde tipos numéricos só de valores positivos até classes que permitem o acesso ao disco. Você pode consultar a documentação da Apple para conhecer mais sobre as classes que deseja em <http://bit.ly/swift-linguagem-guiadereferencia>

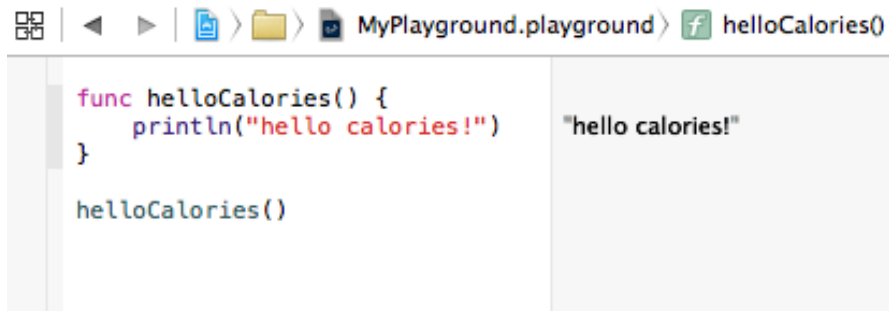
Podemos definir funções novas, com o uso de `func`, e invocá-las:

```
func helloCalories() {
    println("hello calories!")
}

helloCalories()
```

Note que o resultado aparece como a impressão da linha onde fizemos o `println`, e não da linha onde invocamos nosso método. Isso porque o

playground está querendo nos mostrar o que aconteceu com cada linha de código quando elas foram executadas.

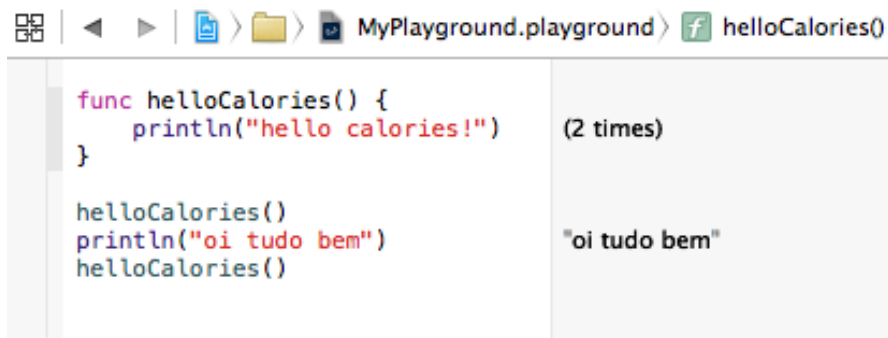


Vamos tentar invocar duas vezes o mesmo método e imprimir outro valor no meio:

```
func helloCalories() {  
    println("hello calories!")  
}
```

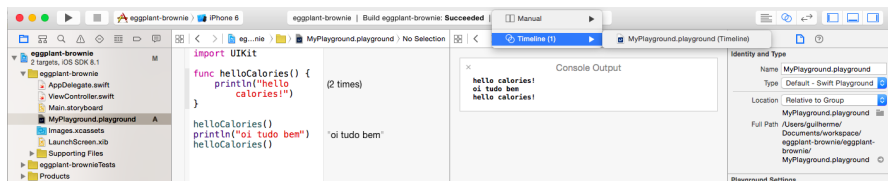
```
helloCalories()  
println("oi tudo bem")  
helloCalories()
```

Observe que, no resultado que temos agora, essa linha foi executada duas vezes.



Por mais que essa visualização nos ajude bastante para entender o resultado de execução de cada linha isoladamente, talvez seja interessante ver o resultado à medida que o código foi executado, ou seja, a saída que foi jogada com `println` à medida que o tempo foi passando, uma *timeline*. É exatamente isso que o assistente *Timeline* que vimos anteriormente faz.

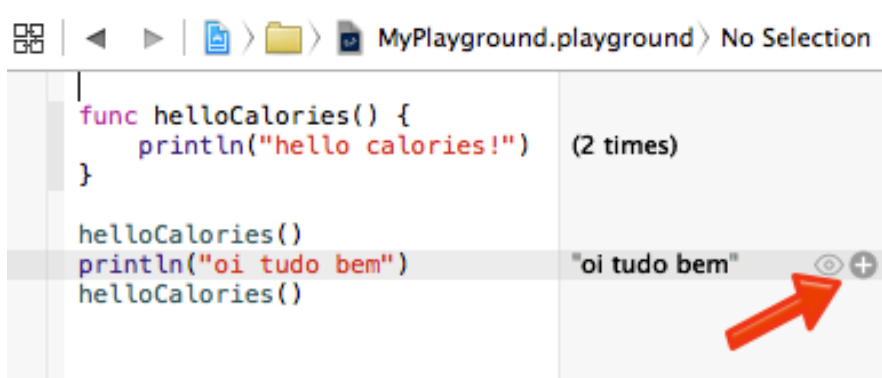
Ao exibi-lo, temos o resultado em ordem do *Timeline*. Se seu Xcode não mostrá-lo no modo assistente, clique onde está escrito *Timeline* como na imagem a seguir e selecione-o:



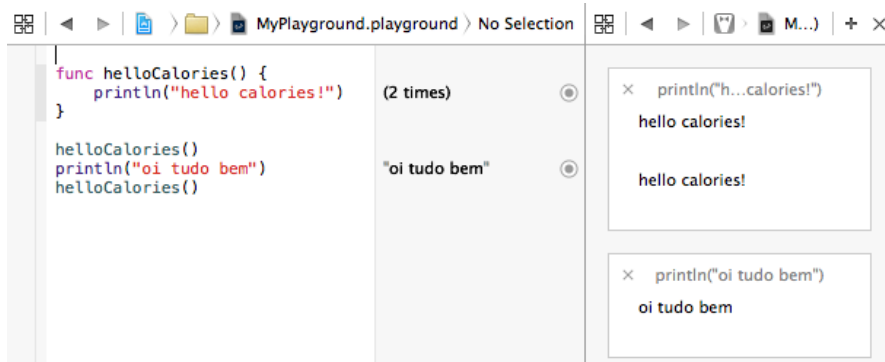
BUG DO ASSISTANT

Caso o assistente não mostre um console vazio, nem mostre o resultado das invocações, você pode voltar ao modo normal e novamente ao assistente. Em algumas versões, o Xcode possui um bug que faz com que, na primeira vez que mudamos para o `Timeline`, ele mostre o console vazio.

Também podemos clicar no ícone circular à esquerda do playground, que se assemelha ao símbolo de `+` quando passamos o mouse por cima dele:



O resultado é que logo abaixo do console conseguimos ver a saída somente desta linha:



As funções podem receber parâmetros, como o nome e o número de calorias ao adicionar um novo item a uma refeição. Quando os recebemos, devemos dizer quais os tipos desses parâmetros:

```
func add(name: String, calories: Double) {
    println("adding \(name) \(calories)")
}
```

```
add("Eggplant", 50.5)
```

3.2 ARRAYS

Até agora estamos trabalhando com tudo em uma única unidade. Não podemos ter duas variáveis com o mesmo nome no mesmo escopo. Como acumular diversos produtos? Diversos elementos do mesmo tipo? Queremos criar um *array*. Podemos criar um array de calorias, por exemplo:

```
let calories = [50.5, 100, 300, 500]
```

Poderíamos opcionalmente declarar o tipo do array explicitamente:

```
let calories:Array<Int> = [ 50.5, 100, 300, 500]
```

Opa, não compila. Claro, um dos valores é `double`, portanto é um array de *doubles*:

```
let calories:Array<Double> = [ 50.5, 100, 300, 500]
```

Somente se o array fosse vazio seria interessante indicar o tipo de array que queremos:

```
let items:Array<Double> = [ ] // Doubles
```

Agora que tenho um array de calorias, com **tipagem implícita**, gostaria de passar por todos os seus itens para somá-los. Como fazer isso? Fazemos um `for` de 0 até 4 (o número de elementos, exclusive):

```
let calories = [ 50.5, 100, 300, 500]
for var i = 0; i < 4; i++ {
    println(calories[i])
}
```

Podemos utilizar a propriedade `count` de um `Array` para fazer esse laço:

```
let calories = [ 50.5, 100, 300, 500]
for var i = 0; i < calories.count; i++ {
    println(calories[i])
}
```

Quando nosso `for` é específico de somar um em um o número do índice, podemos usar o atalho que passa entre 0 e o 3, inclusive:

```
let calories = [ 50.5, 100, 300, 500]
for i in 0...3 {
    println(calories[i])
}
```

Se utilizarmos `calories.count`, temos que tomar cuidado. Esse valor será 4 e teremos um erro, afinal estaríamos iterando da posição 0 a 4 (5 elementos). Queremos, portanto, `calories.count - 1`:

```
let calories = [ 50.5, 100, 300, 500]
for i in 0...(calories.count - 1) {
    println(calories[i])
}
```

Que código feio! Mesmo extraindo uma variável ainda não parece ser um código ideal em nossa situação:

```
let calories = [ 50.5, 100, 300, 500]
let total = calories.count - 1
for i in 0...total {
    println(calories[i])
}
```

Quando desejamos passar por todos os elementos de um `Array`, podemos utilizar o `for in`:

```
let calories = [ 50.5, 100, 300, 500]
for c in calories {
    println(c)
}
```

Por fim, uma função pode retornar algo, como o total de calorias contidas em um `Array` de calorias:

```
func allCalories(calories: Array<Double>) {
    var total = 0
    for c in calories {
        total += c
    }
    return total
}
```

```
allCalories([ 10.5, 100, 300, 500])
```

Mas precisamos dizer qual o retorno da função. Como ela devolve um `Double`, será `-> Double`:

```
func allCalories(calories: Array<Double>) -> Double {
    var total = 0
    for c in calories {
        total += c
    }
    return total
}
```

```
allCalories([ 10.5, 100, 300, 500])
```

Contudo, ainda tem algo de errado. O que não compila nesse código? A variável `total` é implicitamente um `Int` e somos avisados disso. Mudamos então para um `Double`:

```
func allCalories(calories: Array<Double>) -> Double {
    var total = 0.0
    for c in calories {
        total += c
    }
    return total
}
```

```
allCalories([ 10.5, 100, 300, 500])
```

Ou definimos a tipagem explícita:

```
func allCalories(calories: Array<Double>) -> Double {
    var total:Double = 0
    for c in calories {
        total += c
    }
    return total
}
```

```
allCalories([ 10.5, 100, 300, 500])
```

Por fim, quando invocamos uma função que retorna algo, podemos utilizar seu retorno diretamente ou aplicá-lo a uma variável:

```
func allCalories(calories: Array<Double>) -> Double {
    var total:Double = 0
    for c in calories {
        total += c
    }
    return total
}
```

```
let totalCalories = allCalories([ 10.5, 100, 300, 500])
println(totalCalories)
```

3.3 BOA PRÁTICA: CUIDADO COM INFERÊNCIA DE TIPOS

É preciso tomar cuidado com a inferência automática de tipos. Um problema clássico ocorre quando dividimos dois `Ints` e o resultado final é um `Int` sem percebermos, achando que estamos dividindo `Doubles`. Preste muita atenção ao usar a inferência, ela ajuda a digitar menos, mas exige mais na hora de utilizá-la.

```
var values = [ 1, 2]

var total = 0
for v in values {
    total += v
}
println(total / values.count) // 1? 1.5?
```

Tome muito cuidado com tipos implícitos. Digitar menos é bom. Mas o mais importante é que nosso programa funcione.

Parece ser por esse motivo que Swift reforça o tipo de retorno ao invocar um método. Enquanto em Scala, uma linguagem muito parecida em diversos pontos com Swift, o retorno pode ser implícito, em Swift ele não pode. O código a seguir não compila pois a função não indicou qual tipo ela retorna. Ela poderia inferir `Int`:

```
func number() {
    return 15 // compilation error
}

var values = [ number(), number()]
```

Como a linguagem não infere, precisamos declarar o tipo de retorno.

```
func number() -> Int {
    return 15
}
```

```
}
```

```
var values = [ number(), number()]
```

3.4 RESUMO

Neste capítulo, vimos como criar um playground e utilizá-lo para testar características da linguagem ou executar pequenos trechos de código Swift.

Vimos como a linguagem cuida da declaração e inicialização de variáveis, como ela se vira com a tipagem implícita e o cuidado que devemos tomar com ela.

Criamos funções que recebem argumentos, que devolvem valores e trabalhamos com `Int`, `Double`, `Bool` e `Strings`. Fizemos algumas operações aritméticas e utilizamos o laço do tipo `for` para iterar por elementos.

Apesar de não termos utilizado aqui, o `if`, `while` e outros, eles se assemelham em muito a outras linguagens como `Java`, `C` e `C#`. Alguns deles veremos neste material, outros são de simples aprendizado e de fácil consulta no guia de referência da linguagem.

CAPÍTULO 4

Swift: mapeando nossos modelos e introdução à Orientação a Objetos

4.1 BOA PRÁTICA: ORGANIZANDO OS DADOS

Toda vez que tenho uma refeição, devo trabalhar com duas variáveis (nome, felicidade) e um array? Se uma função deseja passar uma refeição para outra função, deve passar esses três argumentos? Toda vez?

E toda função que depende de uma refeição deve ficar jogada pelos cantos do nosso programa? O cálculo de calorias total, o cálculo de felicidade por caloria? Cada coisa jogada em um canto?

Orientação a Objetos apresenta uma solução para esses problemas ao

agrupar os dados e comportamentos ligados em um único lugar, uma única classe.

Swift oferece também características de uma linguagem orientada a objetos. Para agruparmos todos os atributos que uma refeição possui, junto com os comportamentos que ela pode ter, criamos uma classe. A classe é a abstração de uma refeição. Atributos da refeição seriam o nome, a felicidade e seus itens. Já comportamentos (métodos) da classe seriam o cálculo de calorias totais, de felicidade por caloria etc.

Vamos limpar todo nosso playground e colocar a classe `Meal`:

```
class Meal {  
  
}
```

BOA PRÁTICA: PARA SABER MAIS DE ORIENTAÇÃO A OBJETOS

Você pode conhecer mais de Orientação a Objetos estudando o conceito com a apostila da Caelum, disponível online em seu site.

O blog da Caelum também oferece diversos artigos sobre o assunto, como o artigo que mostra como evitar sequências de `ifs` em Orientação a Objetos: <http://bit.ly/como-nao-aprender-oo>

Portanto, no nosso caso, toda refeição terá um nome e um nível de felicidade, os quais chamamos em Swift de propriedades armazenadas:

```
class Meal {  
    var name = "Eggplant Brownie"  
    var happiness = 5  
}
```

Podemos agora criar uma refeição, acessar e alterar seus valores. Para criá-la, usamos seu construtor, seu inicializador, que se assemelha à chamada de uma função (ou método) com o nome igual ao do tipo da classe:

```
let brownie = Meal()  
println(brownie.name)
```

```
println(brownie.happiness)

brownie.happiness = 3
println(brownie.happiness)
```

<pre>1 class Meal { 2 var name = "Eggplant Brownie" 3 var happiness = 5 4 } 5 6 let brownie = Meal() 7 println(brownie.name) 8 println(brownie.happiness) 9 10 brownie.happiness = 3 11 println(brownie.happiness) 12</pre>	<pre>{name "Eggplant Brownie" happin... "Eggplant Brownie" "5" {name "Eggplant Brownie" happin... "3"</pre>
---	--

O que estamos fazendo aqui é definindo uma variável chamada `brownie` que **referencia** um objeto. Ele “aponta”, referencia o objeto que está criado na memória.

Note que, apesar de `brownie` ser uma constante, podemos trocar os valores dentro do objeto. Como os valores foram declarados como `vars`, eles puderam ser alterados; é a referência ao objeto que não pode ser trocada. O código a seguir tenta alterar nossa referência de uma constante, apontando agora para uma nova refeição, e falha:

```
let brownie = Meal()
println(brownie.name)
println(brownie.happiness)

brownie.happiness = 3
println(brownie.happiness)

brownie = Meal() // error!
```

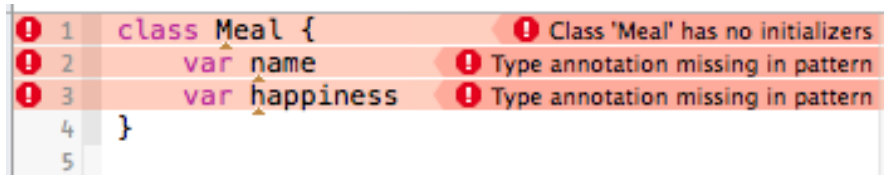
Nesse caso, o que fizemos foi pegar aquela variável que referencia um objeto, e fazê-la passar a referenciar (apontar) para outro objeto. Agora ninguém

mais referencia o primeiro objeto que criamos.

Mas, claro, nem toda refeição será um brownie de berinjela. Não parece fazer sentido que, toda vez que criarmos um objeto do tipo `Meal`, ele venha automaticamente preenchido como tal brownie. Vamos remover o valor inicial e verificar o que acontece... temos três erros em nossa classe!

```
class Meal {  
    var name  
    var happiness  
}
```

Para vermos os erros que aconteceram, podemos clicar na exclamação vermelha que fica à esquerda da linha que ocorreu. A linha inteira ficará vermelha:



Dois destes erros são fáceis de resolver: antes o compilador sabia o tipo das variáveis, pois inferia os mesmos através de suas inicializações, então adicionamos as tipagens explícitas:

```
class Meal {  
    var name: String  
    var happiness: Int  
}
```

Ficamos agora com um único erro, o compilador diz que a classe `Meal` não possui um inicializador. O que está acontecendo? Lembre-se que, em Swift, a não existência de valor (em geral, o `null` em outras linguagens) é um caso importante o suficiente para ter que ser declarado por explícito. Não queremos erros do tipo do acesso a `null` em tempo de execução, por isso a

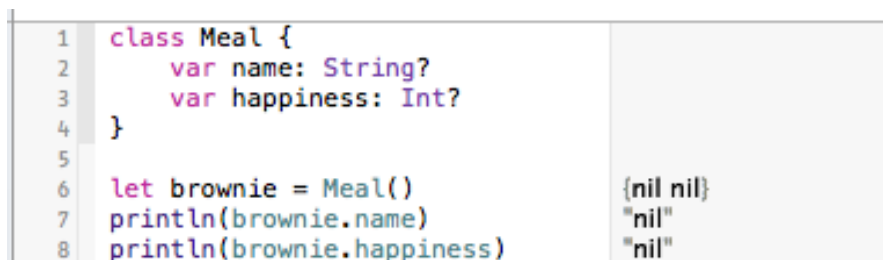
linguagem coloca barreiras para evitar a existência deles. Se o desenvolvedor desejar a existência de um `null`, ele deve ser responsável por marcar aquele campo como tal, um campo que **é opcional**, que pode ter um valor nulo.

```
class Meal {  
    var name: String?  
    var happiness: Int?  
}
```

Repare que a impressão de um nome e do nível de felicidade mostra agora os valores `nil` (o `null` em Swift):

```
class Meal {  
    var name: String?  
    var happiness: Int?  
}
```

```
let brownie = Meal()  
println(brownie.name)  
println(brownie.happiness)
```



The image shows a code editor with Swift code on the left and its output on the right. The code defines a `Meal` class with optional `String?` and `Int?` properties, creates an instance `brownie`, and prints its `name` and `happiness` properties. The output shows `{nil nil}` for the object and `"nil"` for both properties.

```
1 class Meal {  
2     var name: String?  
3     var happiness: Int?  
4 }  
5  
6 let brownie = Meal()  
7 println(brownie.name)  
8 println(brownie.happiness)
```

{nil nil}
"nil"
"nil"

E se colocarmos um valor dentro de nossa refeição, como fica a saída?

```
let brownie = Meal()  
brownie.name = "Eggplant brownie"  
brownie.happiness = 3  
  
println(brownie.name)  
println(brownie.happiness)
```

<pre> 1 class Meal { 2 var name: String? 3 var happiness: Int? 4 } 5 6 let brownie = Meal() 7 brownie.name = "Eggplant brownie" 8 brownie.happiness = 3 9 10 println(brownie.name) 11 println(brownie.happiness) -- </pre>	<pre> {nil nil} {{Some "Eggplant brownie"} nil} {{Some "Eggplant brownie"} {Som "Optional("Eggplant brownie)" "Optional(3)" </pre>
---	--

Ele deixou claro que o que temos agora não é mais uma `String` ou um `Int`, mas sim um `Optional[String]` ou um `Optional[Int]`. O que o compilador faz é: se você deseja uma variável com valor opcional, marque com o `?`. Nesse caso, seu valor inicial é `nil` e, ao colocar algo lá dentro, ele passa a valer `Optional(valor colocado)`.

Observe que podemos deixar explícito o uso do `Optional` ao atribuir valor a uma variável opcional, como no código a seguir:

```

var name:String?
name = Optional("Zucchini muffin")

```

Mas podemos deixar implícita a transformação de um valor para seu tipo `Optional`, como em:

```

var name:String?
name = "Zucchini muffin"

```

As duas abordagens produzem o mesmo resultado para o compilador, uma vez que, a partir do momento que a variável for declarada com `String?`, ela será tratada dessa forma em ambos os casos.

Temos algo de estranho ainda: não desejamos imprimir `Optional("Zucchini muffin")`, mas sim somente o nome, a `String`.

```

var name:String?
name = "Zucchini muffin"
println(name)

```

Pior ainda, se tentarmos imprimir o tamanho de nossa `String` usando a função `countElements`, percebemos que para uma declaração normal, ela funciona:

```
var name = "brownie"
println(countElements(name)) // 7
```

Mas não funciona para o `Optional`, afinal o método recebe `String`, e não `String?`:

```
var name:String?
name = "brownie"
println(countElements(name)) // error: type 'String?'
                                // does not conform to protocol
                                // '_CollectionType'
```

Acontece que um `Optional` não é do tipo que queremos, desejamos extrair o que está lá dentro, tanto na hora da impressão quanto no momento de usar o conteúdo de nossa `String?`. Mas o compilador não é bobo, ele quer nos ajudar e dizer: desenvolvedor, ao tentar usar essa variável, você tem na verdade um `Optional`, tem um risco aí, talvez seja algo `nil`. Tem certeza? Desenvolvedor, se você acessá-la sem verificar se é válida, é capaz de seu programa parar completamente - pior ainda, ele com certeza vai parar caso seja `nil` e você chame um método dele.

Vamos dizer ao compilador que sabemos o que estamos fazendo. Nós sabemos que o que está lá dentro é algo não nulo, e que assumimos o risco utilizando o caractere `!`:

```
var name:String?
name = "brownie"
println(countElements(name!)) // 7
```

Perfeito, mas isso é arriscado, olhe o que acontece caso seja `nil`, o caso que o compilador tentou a todo custo evitar:

```
var name:String?
println(countElements(name!)) // fatal error
```

CUIDADO COM BUGS DO COMPILADOR DO XCODE

Como Swift é uma linguagem nova, o compilador utilizado pelo Xcode pode se confundir com algumas coisas. Nos exemplos feitos a partir deste capítulo, tome cuidado: caso a mensagem de erro apareça em uma linha diferente daquela esperada, teste o caso que está interessado isoladamente. Fazer cada um dos testes de maneira isolada evita deixar o compilador perdido. No playground, isso ocorre mais frequentemente, uma vez que ele utiliza informações de debug para imprimir os resultados do lado direito de nossa IDE.

Não queremos que isso aconteça, portanto o mais comum é verificar se o campo é não nulo, e só então utilizá-lo:

```
if (name != nil) {
    println(countElements(name!)) // 7
} else {
    println("empty")
}
```

No Swift, os parênteses do `if` (e dos laços) é opcional, portanto podemos fazer:

```
if name != nil {
    println(countElements(name!)) // 7
} else {
    println("empty")
}
```

Como estamos utilizando nossa `String` mais de uma vez dentro do código, podemos querer fugir do uso do `!` ao fazer o `if` e já declarar uma variável com o valor real ao mesmo tempo:

```
var name:String?
if let n = name {
    println(countElements(n)) // 7
} else {
```

```
        println("empty")
    }
```

Mas sejamos sinceros. Nesse caso, nós temos certeza de que existe um valor na nossa `String` e, apesar de a declararmos como opcional, gostaríamos de dizer ao compilador que temos certeza de que ela tem um valor válido, usando o `!` na declaração da variável:

```
var name:String!
name = "Eggplant Brownie"

println(countElements(name)) // 7
```

Lembre-se de tomar muito cuidado toda vez que usar o `!`, seja para acessar o valor de uma variável opcional, seja ao declarar uma variável opcional quando você tem certeza de que ela será inicializada.

Se tentássemos usá-lo sem o valor inicializado, a responsabilidade seria do programador e o erro seria fatal:

```
var name:String!

println(countElements(name)) // fatal error
```

E toda essa história de uma linguagem tratar um valor opcional (`Optional`) com tanto carinho, como fica nos casos de invocação de métodos que retornam valores opcionais? Vamos definir uma `String` com o valor `5` e transformá-la em um `Int`, qual o tipo retornado?

```
let happiness = "5"
println(happiness.toInt())
```

O resultado é um `Optional<Int>`, afinal pode ser que dê um erro e o valor aí dentro de nossa `String` não seja um número. Repare que o `Optional` pode ser usado como alternativa no retorno de um método para dizer que o processo foi um sucesso ou não. É um *tradeoff* que os implementadores da linguagem escolheram na hora de implementar o método `toInt` da `String`.

4.2 BOA PRÁTICA: GOOD CITIZEN (EVITE NULOS) E O INIT

Sempre que criamos uma refeição, precisamos de um nome e de um nível de felicidade. Essas não são características opcionais. Diversas linguagens orientadas a objeto favorecem e pregam a “boa prática” (ironia intencional) de deixar tudo mutável através de *getters* e *setters*, propriedades ou qualquer outro nome dado a uma variável membro (propriedade) que é mutável e inicializada com zero ou nulo.

O perigo de uma variável não inicializada é simples: você tem um objeto que pode não estar preparado para ter um método chamado. O padrão *Good Citizen* diz que todo objeto, assim que criado, deve estar pronto para ter todos os seus métodos executados. Não existe “não invoque enquanto” ou “invoque somente após”. Com isso evitamos, por exemplo, os diversos *null pointers* ou *fatal errors* que podemos tomar em nossas aplicações: todos os valores estão preenchidos, nenhum está vazio!

Swift reforça isso com a utilização de variáveis opcionais explícitas: se você quiser que ela seja opcional, deve explicitar, e deve também, quando acessá-la, explicitar que sabe o que está fazendo. São dois passos que um desenvolvedor precisa tomar antes de fazer uma bobagem.

CODE SMELL: COMO NÃO APRENDER ORIENTAÇÃO A OBJETOS: GETTERS E SETTERS

O blog da Caelum possui um post de Paulo Silveira sobre como não aprender Orientação a Objetos, e os problemas da utilização de *getters* e *setters*:

<http://bit.ly/encapsulamentoGetterSetter>

4.3 NOSSO GOOD CITIZEN

Devemos obrigar todos os programadores que desejam instanciar *Meal* a passar os dois argumentos. É o que definimos no nosso inicializador, o construtor, e alteramos nosso código para garantir isso, não queremos opcionais!

```
class Meal {
    var name:String?
    var happiness:Int?
    init(name: String, happiness: Int) {
    }
}
```

Podemos instanciar e mostrar que nosso `Meal` sempre tem valor, e nunca aceita `nil`. Podemos ser felizes com um `Meal`, que ele nunca vai dar um fatal error (ou equivalente a um `Null pointer`):

```
let brownie = Meal("Eggplant brownie", 5)

println(brownie.name)
println(brownie.happiness)
```

Mas o compilador reclama. Ao inicializarmos um objeto durante a construção, é obrigatória a passagem do nome dos parâmetros, ficando claro o que é o quê:

```
let brownie = Meal(name: "Eggplant brownie", happiness: 5)

println(brownie.name)
println(brownie.happiness)
```

Mas se recebemos o nome e a felicidade no construtor, não precisamos dos opcionais:

```
class Meal {
    var name:String
    var happiness:Int
    init(name: String, happiness: Int) {
    }
}
```

Agora o Xcode reclama de que não inicializamos as variáveis que não são opcionais. Desejamos ser um bom cidadão, e, se você possuir uma referência para um `Meal`, pode ter certeza de que tudo que está lá dentro está bem configurado, com todos os valores preenchidos. Portanto, em nosso construtor, atribuímos os valores dos parâmetros para nossas propriedades. Para

isso, referenciamos o próprio objeto que está sendo construído (ou que já foi construído) através da referência `self`:

```
class Meal {
    var name:String
    var happiness:Int
    init(name: String, happiness: Int) {
        self.name = name
        self.happiness = happiness
    }
}
```

Agora sim, nosso código compila e podemos ver que ele já tem valores válidos logo em sua inicialização. Podemos também conferir que a execução de nosso código não permite a passagem de nulo:

```
let brownie = Meal(name: nil, happiness: 5) // compile error
```

Temos nosso `Good Citizen`, um objeto do qual, quando temos uma referência, temos certeza de que duas propriedades estão bem definidas e podemos utilizá-lo.

Seguindo os mesmos princípios, criamos nosso `Item`:

```
class Item {
    var name:String
    var calories:Double
    init(name: String, calories: Double) {
        self.name = name
        self.calories = calories
    }
}
```

```
class Meal {
    var name:String
    var happiness:Int
    init(name: String, happiness: Int) {
        self.name = name
        self.happiness = happiness
    }
}
```

Mas queremos que nossa refeição possua diversos itens. Usemos, então, um array de itens:

```
class Item {
    var name:String
    var calories:Double
    init(name: String, calories: Double) {
        self.name = name
        self.calories = calories
    }
}

class Meal {
    var name:String
    var happiness:Int
    var items
    init(name: String, happiness: Int) {
        self.name = name
        self.happiness = happiness
    }
}
```

Mas isso não faz sentido. Queremos que a variável `items` seja um array vazio de itens:

```
class Item {
    var name:String
    var calories:Double
    init(name: String, calories: Double) {
        self.name = name
        self.calories = calories
    }
}

class Meal {
    var name:String
    var happiness:Int
    var items = Array<Item>()
    init(name: String, happiness: Int) {
```

```
        self.name = name
        self.happiness = happiness
    }
}
```

Além de propriedades e construtores, uma classe pode ter métodos, como um que nos ajude a calcular o total de calorias de uma refeição:

```
class Item {
    var name:String
    var calories:Double
    init(name: String, calories: Double) {
        self.name = name
        self.calories = calories
    }
}
```

```
class Meal {
    var name:String
    var happiness:Int
    var items = Array<Item>()
    init(name: String, happiness: Int) {
        self.name = name
        self.happiness = happiness
    }

    func allCalories() -> Double {
        var total = 0.0
        for i in items {
            total += i.calories
        }
        return total
    }
}
```

Podemos criar agora um novo item:

```
let brownie = Meal(name: "Eggplant brownie", happiness: 5)
let item1 = Item(name: "brownie", calories: 115)
let item2 = Item(name: "vegan cream", calories: 40)
```

Mas queremos adicioná-los nos itens do brownie. Por isso, chamamos o método `append`:

```
let brownie = Meal(name: "Eggplant brownie", happiness: 5)
let item1 = Item(name: "brownie", calories: 115)
let item2 = Item(name: "vegan cream", calories: 40)
brownie.items.append(item1)
brownie.items.append(item2)
```

Como as variáveis só foram utilizadas para invocar o método `append`, podemos invocá-lo direto, passando como parâmetro a referência ao objeto que acaba de ser inicializado:

```
let brownie = Meal(name: "Eggplant brownie", happiness: 5)
brownie.items.append(Item(name: "brownie", calories: 115))
brownie.items.append(Item(name: "vegan cream", calories: 40))

println(brownie.allCalories())
```

O resultado do `println` é `155.0`, como esperado.

BOA PRÁTICA: ENCAPSULAMENTO

Encapsulamento é esconder como as coisas são feitas e permitir acesso a uma interface que executa as tarefas pedidas. No nosso caso, deixamos de acessar diretamente nossos itens e passamos a perguntar à refeição qual o total de calorias.

Encapsular vai muito além de uma mera definição de método, devemos controlar o escopo de acesso de nossas variáveis, métodos e até mesmo classes e pacotes. Veremos outras características ligadas ao encapsulamento ao decorrer do material, e existe uma leitura abrangente sobre o tema no blog da Caelum em:

- <http://bit.ly/revisitandoOO>
- <http://bit.ly/encapsulamentoScala>
- <http://bit.ly/encapsulamentoGetterSetter>
- <http://bit.ly/encapsulamentoEModificadores>

4.4 CODE SMELL (MAL CHEIRO DE CÓDIGO): OPCIONAL!

O uso do opcional com o `!` no Swift como propriedade indica que uma variável terá um valor em tempo de execução mas que o compilador não sabe disso, por isso o programador forçou o `!` goela abaixo na definição da variável. Mas como pode ser que o compilador não saiba que vai ter valor?

Isso acontece comumente, já que a variável utilizada não será inicializada no construtor. Como vimos como `Good Citizen`, a inicialização no construtor evita problemas em tempo de execução; no caso do Swift ele evita *fatal errors*. Portanto, a regra geral é usar sempre valores válidos, evitando `Optional`, e quando usar `Optional`, tentar usar sempre `?` com `if` para garantir o valor; por fim, usar `!`, nessa ordem de preferência.

Veja que os exemplos a seguir todos causam um erro em execução e um *crash* de sua aplicação, pois o desenvolvedor cometeu o erro de não checar o valor opcional. O código do `Meal` está protegido, mas antes de invocar o construtor, ao extrair o valor da variável `name`, o desenvolvedor comete o erro:

```
var name:String?  
let brownie = Meal(name: name!, happiness: 5)
```

Ou ainda em:

```
var name:String!  
let brownie = Meal(name: name, happiness: 5)
```

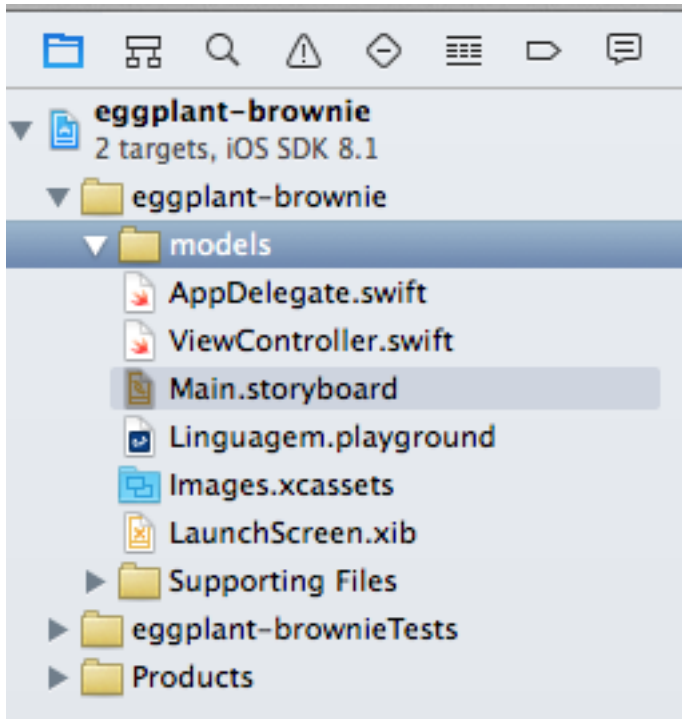
4.5 RESUMO

Vimos até aqui como definir uma classe, suas propriedades, métodos, inicializadores, como instanciá-la e como invocar seus métodos. Vimos também boas práticas e cuidados a serem tomados durante a utilização de variáveis opcionais e obrigatórias. Por fim, colocamos o método em seu devido lugar: um método fica em quem sabe lidar com determinados dados, Orientação a Objetos agrupa dados e comportamentos. No próximo capítulo, aplicaremos tudo isso ao nosso projeto.

CAPÍTULO 5

Projeto: organizando modelos com OO e arquivos em grupos

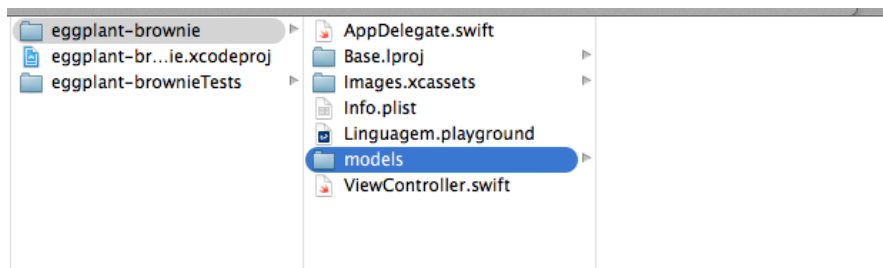
Voltando ao código do projeto, podemos criar um novo arquivo que contenha nossas classes de modelo. Mas os arquivos facilmente ficarão bagunçados. Para agrupá-los, o Xcode possui uma funcionalidade de criar grupos. Clicando no projeto, vamos no menu ‘File’, ‘New Group’ e chamamos ele de `models`. Nosso projeto fica assim:



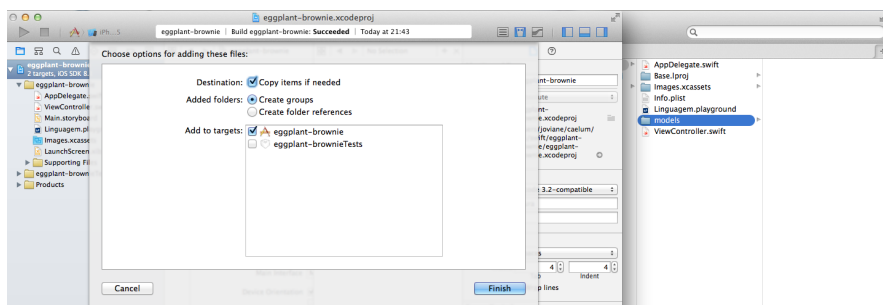
O problema é que no sistema operacional ele não cria diretório nenhum e, de repente, temos diversos arquivos no mesmo diretório no sistema operacional (e no nosso repositório).

5.1 BOA PRÁTICA: CRIE UM DIRETÓRIO POR GRUPO

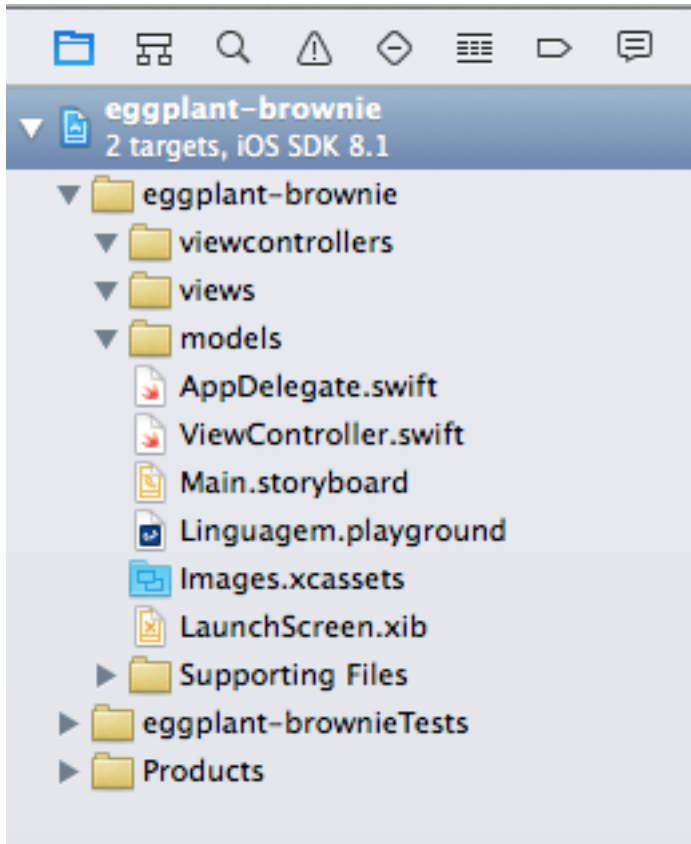
Removemos esse grupo usando o delete. Antes de criar um grupo, fazemos um diretório com o mesmo nome no sistema operacional. Faremos isso criando o diretório `models` dentro do `eggplant-brownie` do nosso projeto:



Agora, arrastamos esse diretório para dentro do nosso projeto na pasta `eggplant-brownie`, e confirmamos, o que automaticamente cria esse grupo. A partir desse momento, ao criarmos os arquivos dentro desse grupo, eles são automaticamente gravados no diretório que criamos, assim sempre teremos um sistema de arquivos refletindo o que aparece dentro do Xcode.



Aproveitamos e criamos nossos diretórios e grupos para `views` e `viewcontrollers`:



Dentro do nosso grupo `models`, criaremos dois arquivos Swift, `Meal.swift` e `Item.swift`. Para isso, clicamos no grupo `models` segurando o control ou com o botão direito e escolhemos `New File...`, `Swift File` e escolhemos o nome `Meal.swift`.

```
class Meal {  
    var name:String  
    var happiness:Int  
    var items = Array<Item>()  
    init(name: String, happiness: Int) {  
        self.name = name
```

```
        self.happiness = happiness
    }

    func allCalories() -> Double {
        println("calculating")
        var total = 0.0
        for i in items {
            total += i.calories
        }
        return total
    }
}
```

Fazemos o mesmo para o `Item`:

```
class Item {
    var name:String
    var calories:Double
    init(name: String, calories: Double) {
        self.name = name
        self.calories = calories
    }
}
```

Mas para que usaremos `var` se tais valores são, por enquanto, imutáveis? Seguindo a boa prática de manter imutável aquilo que não deve mudar, redefinimos todos os campos, exceto o array como `let`:

```
class Meal {
    let name:String
    let happiness:Int
    var items = Array<Item>()
    init(name: String, happiness: Int) {
        self.name = name
        self.happiness = happiness
    }

    func allCalories() -> Double {
        println("calculating")
    }
}
```

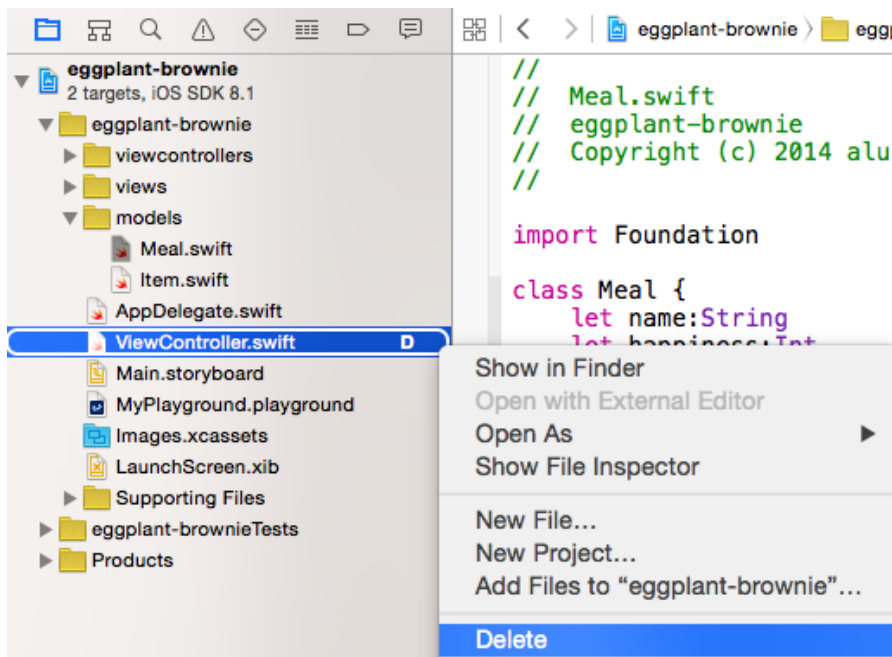
```
        var total = 0.0
        for i in items {
            total += i.calories
        }
        return total
    }
}

class Item {
    let name:String
    let calories:Double
    init(name: String, calories: Double) {
        self.name = name
        self.calories = calories
    }
}
```

Qual o motivo de não colocar o `array` como constante? Acontece que ele poderá ter seu valor alterado com o passar do tempo. Pretendemos adicionar itens nele através do método `append` que vimos antes. Portanto, o `array` ainda é mutável.

5.2 MOVENDO ARQUIVOS NO SISTEMA OPERACIONAL

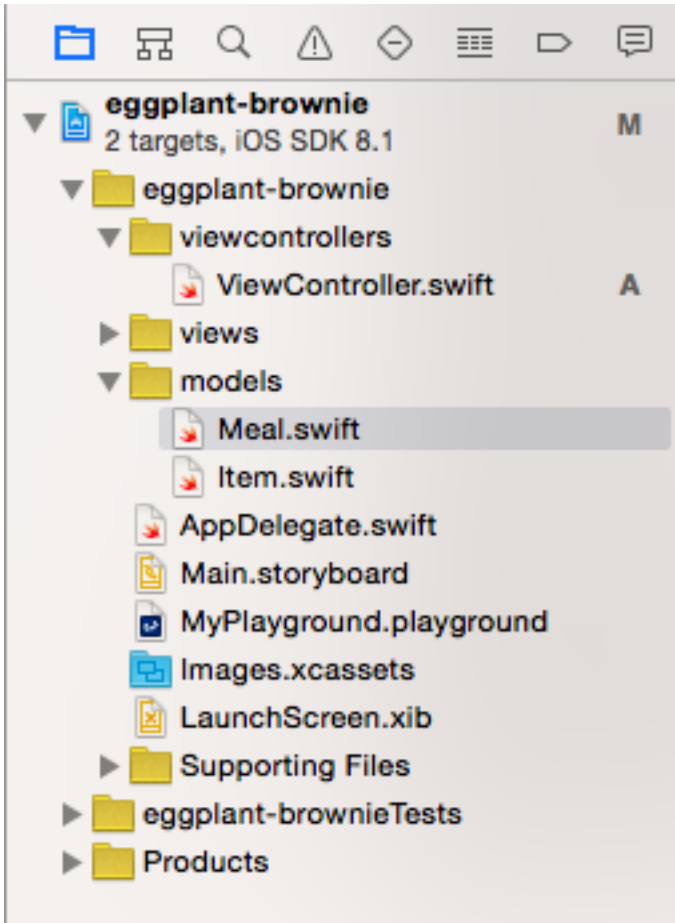
Vamos mover por fora nosso arquivo `ViewController`. Primeiro, no **sistema operacional**, o movemos para o diretório `viewcontrollers`. Ao voltar ao `storyboard`, percebemos que ele está marcado em vermelho, como removido. Clicamos com o botão direito e confirmamos a operação de `delete`:



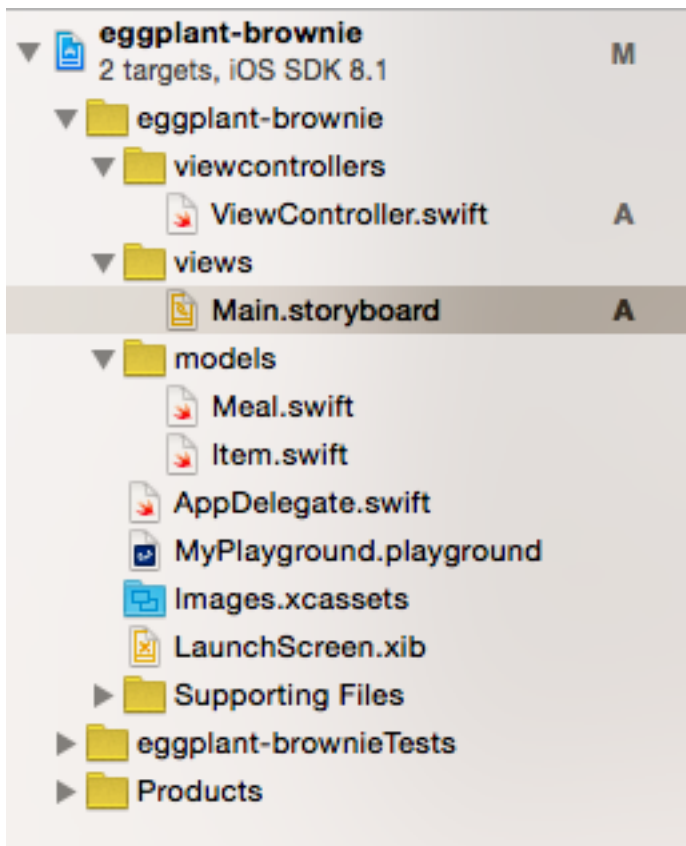
Git

A marcação vermelha com a letra D só aparece caso o seu projeto esteja configurado para utilizar um repositório do tipo Git. Por padrão o Xcode cria seu projeto dessa maneira, mas ele pode não aparecer para você caso tenha desselecionado essa opção durante a criação de seu projeto.

Agora o arrastamos do sistema operacional para dentro do nosso projeto, dentro do grupo que queremos, o `viewcontrollers`. O Xcode pergunta se desejamos criar grupos para diretórios. Como dessa vez estamos arrastando somente um arquivo, podemos clicar `Ok` sem medo:



Já o `Main.storyboard` está dentro diretório `Base.lproj`; vamos movê-lo para o `views`. Executamos o mesmo processo de no Xcode remover o arquivo e efetuar o `drag and drop`:



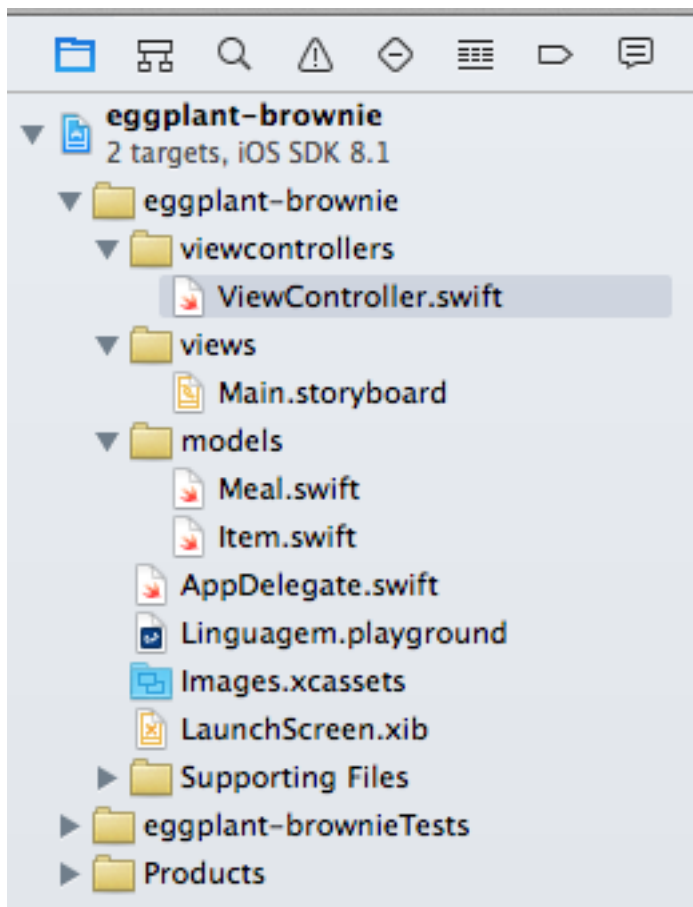
Como mudamos os diretórios de diversos arquivos que estavam conectados via texto em configuração, precisamos garantir que todas as conexões ainda estão ok.

Abrimos o storyboard, clicamos no ícone amarelo de `ViewController` e conferimos se a classe de nosso `ViewController` ainda é a certa:



Se estiver certo, clicamos na seta para ver o código. Se o código não estiver conectado, use o `assistant editor` para conectar novamente o botão e as caixas de texto.

Nosso projeto fica então:



5.3 APLICANDO O BÁSICO DE ORIENTAÇÃO A OBJETOS

Agora devemos utilizar o que aprendemos de OO em nosso projeto. De volta ao nosso controller:

```
@IBAction func add() {  
    let name = nameField.text;  
    let happiness = happinessField.text;
```

```
println("eaten: \(name) \(happiness)!")
}
```

Podemos instanciar um novo `Meal`:

```
@IBAction func add() {
    let name = nameField.text
    let happiness = happinessField.text
    let meal = Meal(name: name, happiness: happiness)
    println("eaten: \(meal.name) \(meal.happiness)")
}
```

Mas o código não compila, o `happiness` deveria ser um número inteiro:

```
@IBAction func add() {
    let name = nameField.text
    let happiness = happinessField.text.toInt()
    let meal = Meal(name: name, happiness: happiness)
    println("eaten: \(meal.name) \(meal.happiness)")
}
```

E ainda não compila: o método `toInt` devolve um opcional, isso fica claro se tentássemos aplicar tipagem explícita:

```
@IBAction func add() {
    let name:String = nameField.text
    let happiness:Int = happinessField.text.toInt()
    let meal = Meal(name: name, happiness: happiness)
    println("eaten: \(meal.name) \(meal.happiness)")
}
```

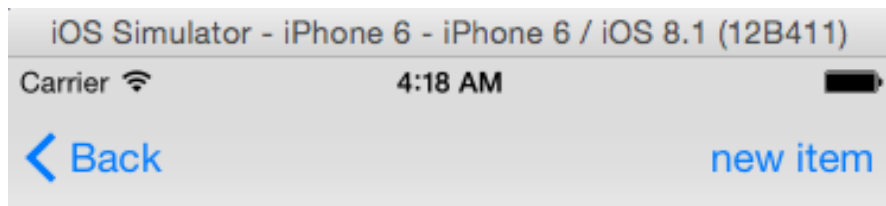
Não estamos aqui para brincadeira, portanto só acessaremos os valores após ter certeza de que eles são válidos: não queremos nossa aplicação quebrando. Extraímos o valor de dentro deles, e removemos a tipagem explícita:

```
@IBAction func add() {
    if nameField == nil || happinessField == nil {
        return
    }
}
```

```
let name = nameField!.text
let happiness = happinessField!.text.toInt()
if happiness == nil {
    return
}

let meal = Meal(name: name, happiness: happiness!)
println("eaten: \(meal.name) \(meal.happiness)")
}
```

No futuro veremos como mostrar uma mensagem de alerta indicando que ocorreu algum erro, uma mensagem educada. Testamos nossa aplicação e vemos que o código funciona.



Name

Happiness

Add

Eggplant Brownie

Zucchini Muffin

Cookie

Coconut oil

Chocolate frosting

Chocolate chip

sundubu

Lembre-se: se por algum motivo desconhecido o campo `happinessField` tivesse um valor que não fosse válido para a transformação via `toInt`, teríamos um `nil` e nada aconteceria.

5.4 RESUMO

Não só sabemos utilizar a IDE e a linguagem, mas já conhecemos diversas boas e más práticas de programação para IOS com Swift. Aprendemos a organizar nossos arquivos em diretórios no sistema operacional e em grupos dentro do Xcode. Vimos também como aplicar os conceitos básicos de Orientação a Objetos ao código que havíamos implementado anteriormente. Reforçamos a validação de valores opcionais para deixar o nosso código menos propício a *fatal errors* e *crashes* da aplicação.

CAPÍTULO 6

Listando as refeições com TableViewController

Nosso próximo passo é listar todas as nossas refeições atuais, permitindo linkar o usuário para uma tela onde ele adiciona uma nova refeição. Esta última já foi criada, mas precisamos ainda aprender a fazer uma tela com uma tabela e a trabalhar com a navegação entre telas, i.e. entre *views*, um conceito muito importante.

O primeiro passo será aprender a criar uma tabela de tamanho fixo e entender o que é necessário fornecermos para criá-la.

Começamos criando um novo projeto no Xcode, temporário, pressionando `Command+Shift+N`. Escolhemos novamente `iOS`, `Single View Application` e damos um nome qualquer como `meal-table`. Dessa vez, após abrirmos o `Main.storyboard`, não colocamos nenhum `label`, que-

remos uma tabela.

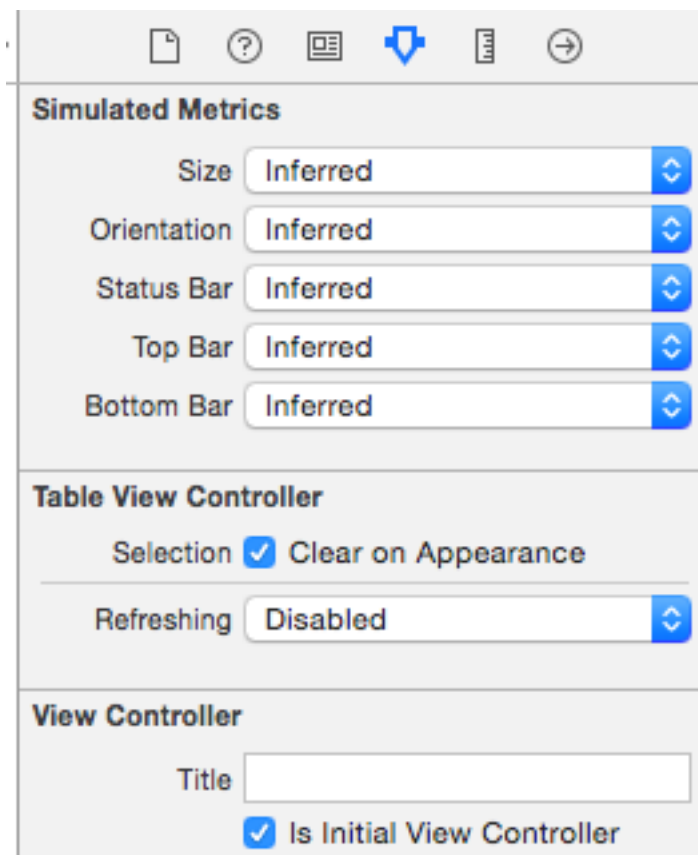
Como é extremamente comum que uma tabela seja o componente único de um `ViewController`, temos um componente já chamado `Table View Controller`, que pode ser criado como um `ViewController` que já possui um único elemento: uma `Table View`.

Com o `storyboard` de nosso projeto aberto, selecionamos nosso `ViewController` padrão e o apagamos usando o clique da direita. Nosso `storyboard` fica vazio. Na lista de componentes escolhemos e arrastamos um `Table View Controller` para nosso `storyboard`.

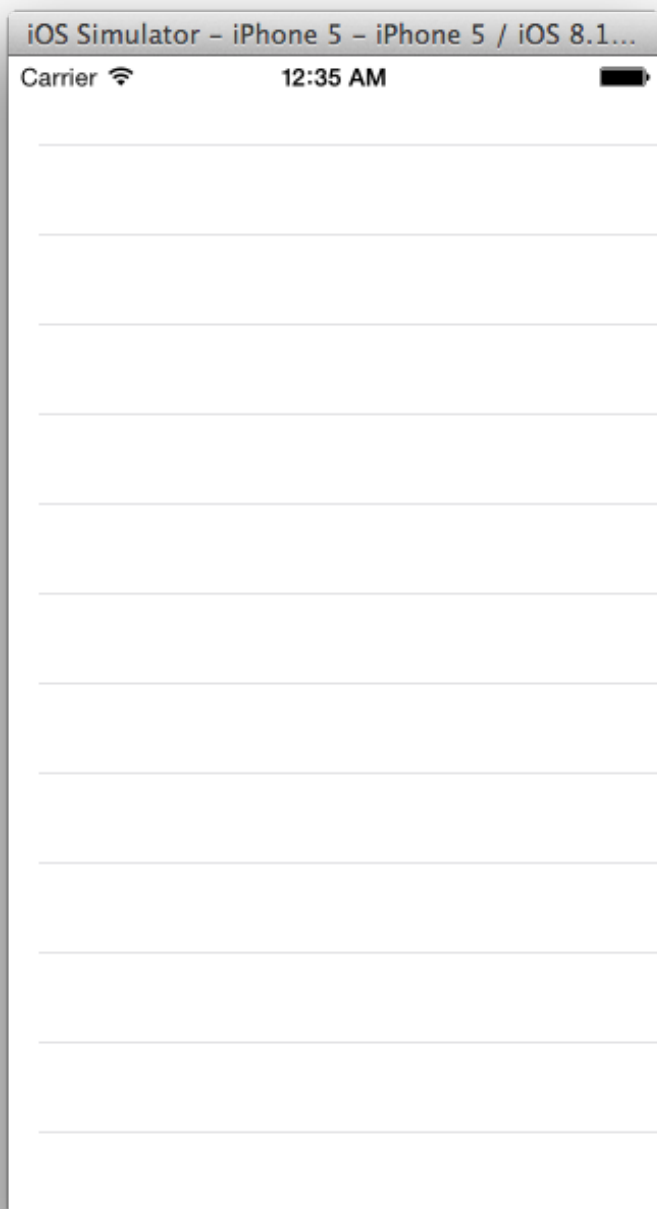
Lembre-se de alterar o tamanho da tela para o iPhone que desejar clicando no `UITableViewController`, ícone amarelo à esquerda, indo nas propriedades e selecionando o `size`.

Mas ao rodar o programa temos uma tela preta, o que aconteceu? Quando deletamos o `ViewController` inicial, o programa perdeu a referência do ponto de entrada da nossa aplicação. O ponto de entrada é indicado por uma seta que aparece no lado esquerdo do nosso `ViewController`.

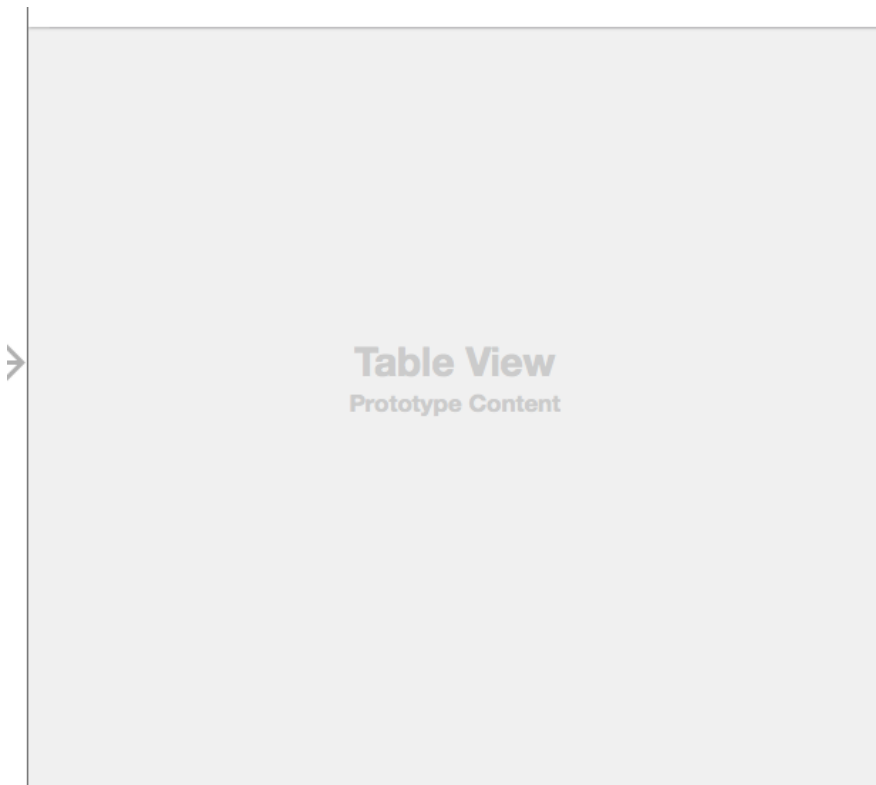
O que fazemos é selecionar o nosso `Table View Controller`, clicando na imagem amarela, ir às propriedades (`Attributes Inspector`), e em `Attributes Inspector` marcar a opção `Is Initial View Controller`:



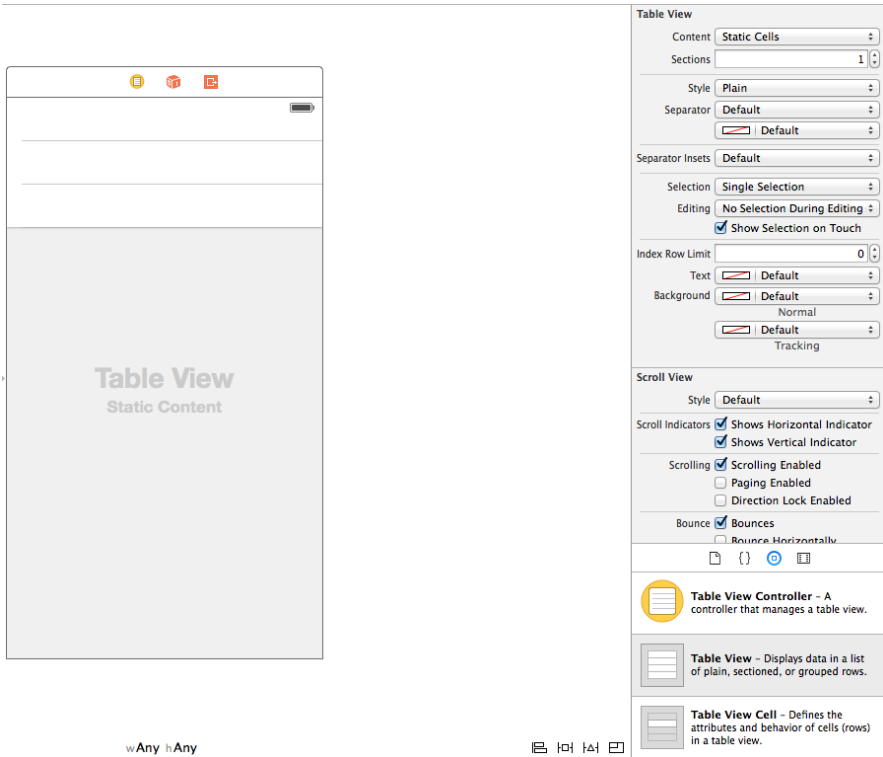
Agora sim rodamos o programa e vemos a tela com nossa tabela:



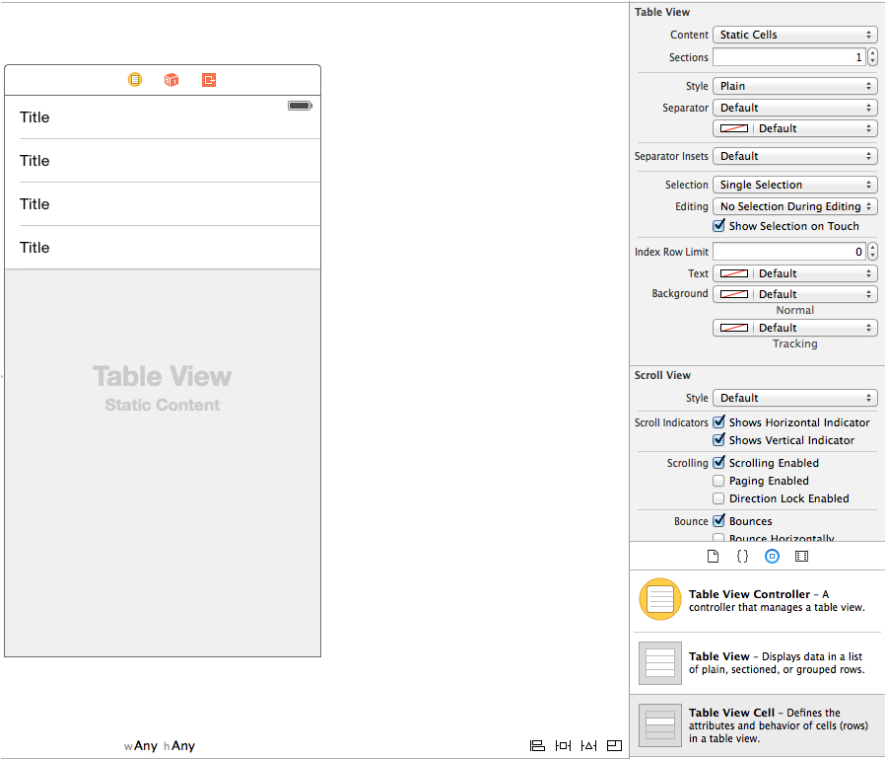
Primeiro fixaremos o número de linhas de nossa tabela selecionando a `Table View`. Confira que ela foi selecionada clicando dentro de onde está escrito `Table View Prototype Content`:



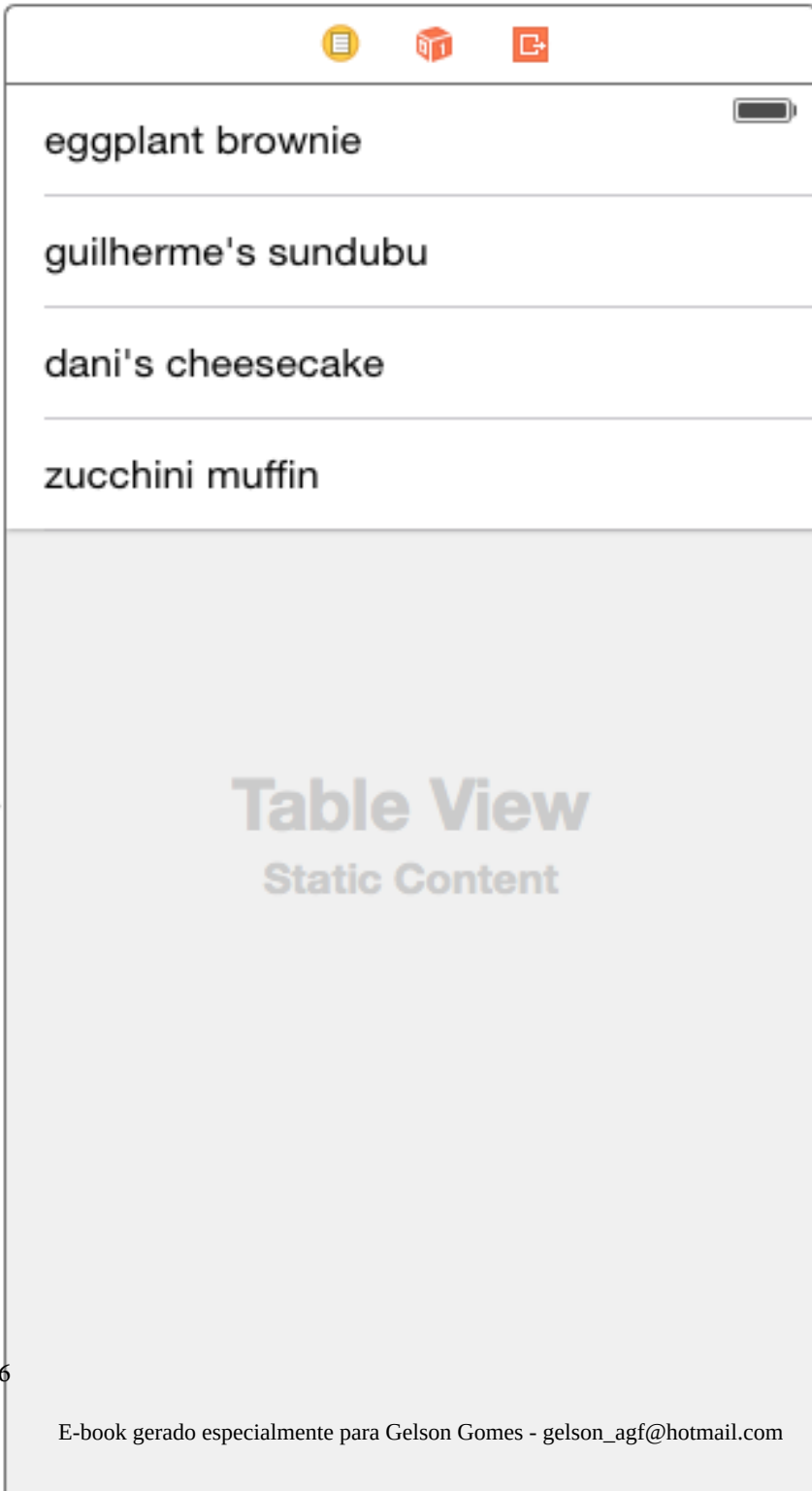
Agora, na barra de propriedades à direita, mudamos o conteúdo para o tipo `Static Cells`. Assim que fazemos a mudança, fica claro que por padrão temos 3 células em nossa tabela. Se desejamos um número fixo de linhas nela, podemos remover células selecionando-as ou adicionar novas células em uma tabela utilizando o componente `Table View Cell`:



Adicionamos a quarta célula arrastando o componente do canto inferior direito para nossa tabela para exemplificar o funcionamento de uma tabela customizada. Agora queremos que cada uma das células contenha um texto básico. Isto é, queremos que o estilo de nossas células seja `basic`. Para isso, selecionamos as quatro células usando o `command click` em cada uma delas. Nas propriedades, alteramos o `style` para `basic` e o resultado são quatro linhas com o texto `Title`, que funciona basicamente como um label tradicional:



Usando o `double click` em cada um dos textos, alteramos seus valores para quatro refeições distintas:



Rodamos o programa e temos nossa tabela!

SELECIONANDO O VIEW CONTROLLER INICIAL

Caso ainda tivéssemos o `View Controller` inicial, poderíamos ter arrastado a seta para o `Table View Controller`, indicando com isso que ele seria o ponto de entrada de nossa aplicação.

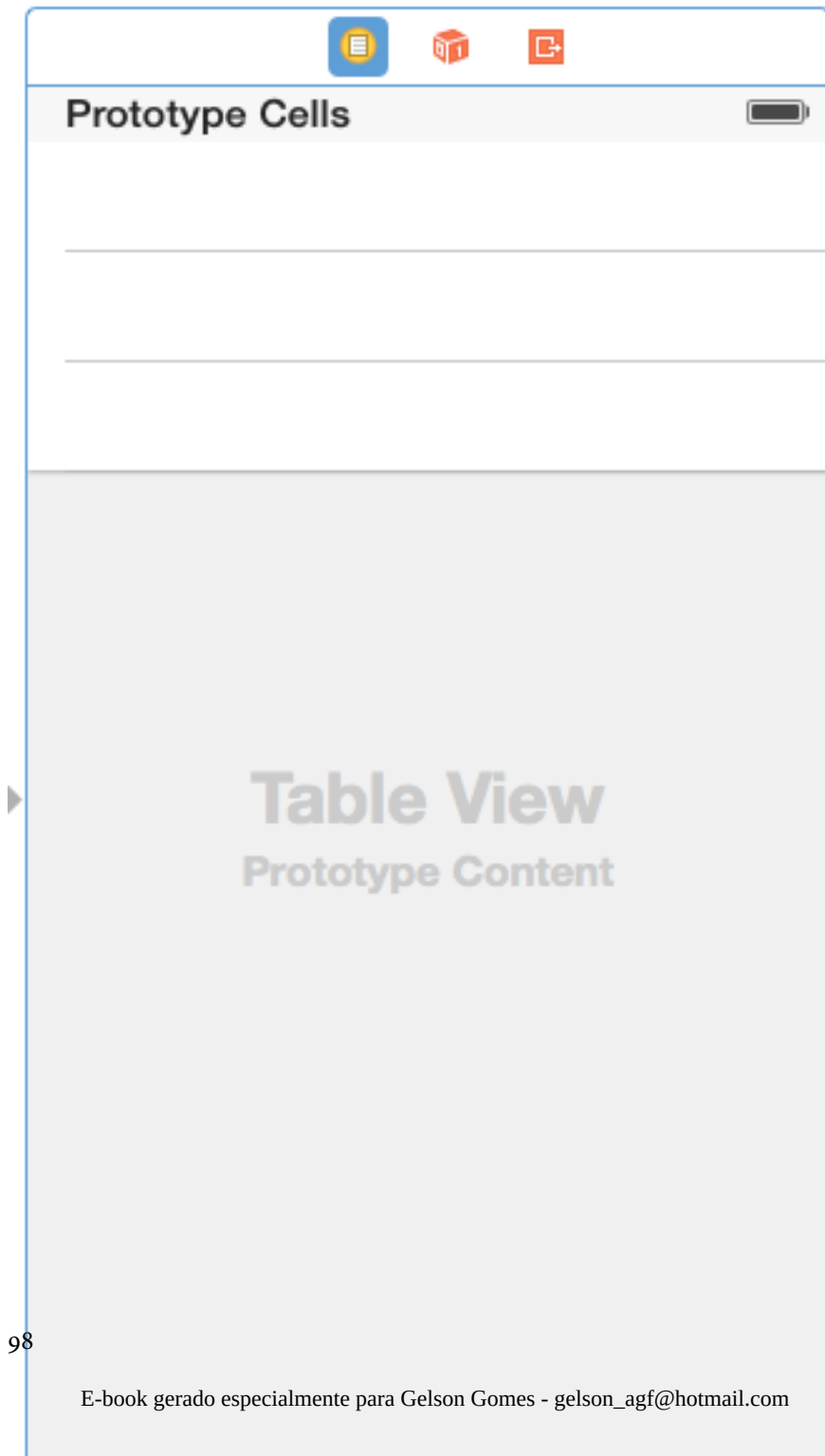
6.1 TABELAS DINÂMICAS USANDO DYNAMIC PROTOTYPES

Mas nem sempre sabemos em tempo de programação quantas linhas desejamos ter em uma tabela e seu conteúdo exato. Em geral, desejamos que a tabela represente um conjunto de dados que temos em memória, algum tipo de `Array`. À medida que modificamos o `Array` desejamos atualizar a tabela para refletir tal mudança.

Para isso, usaremos um outro tipo de tabela, a baseada em `Dynamic Prototypes`. Criamos um novo projeto com `Single View Application` como feito anteriormente. Podemos chamá-lo de `dynamic-meal-table`. Arrastamos um `Table View Controller` e arrastamos a seta para que ele possa ser nossa tela inicial. Deletamos também o `View Controller` original.

Escolhendo agora nossa tabela, conferimos a opção `Content` na barra de propriedades à direita, que já deve estar selecionada como `Dynamic Prototypes`.

Agora queremos conectar nosso código `ViewController.swift` com esse novo `controller` que criamos no `storyboard`. Como fazer isso? Selecionando o `Table View controller`, clicando em seu ícone amarelo no tipo da janela do `table view` à direita:



Podemos no `Identity Inspector` (terceiro ícone da esquerda pra direita na barra de propriedades) escolher quem é o código que identifica esse controller no nosso storyboard. Estamos conectando a view inteira com a nossa classe Swift.

Estranhamente, ao começar a digitar `ViewController` (o nome de nossa classe), o Xcode não a sugere, por quê?

Acontece que nossa classe não representa um `UITableViewController`, mas somente um `ViewController` comum. Alteramos nosso código para herdar de `UITableViewController` já removendo o segundo método que não nos interessa:

```
class ViewController : UITableViewController {  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
    }  
  
}
```

Mas para nosso teste, alteraremos o `viewDidLoad` para que, ao ter a view carregada, mostremos uma mensagem de sucesso:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    println("view did load")  
}
```

HERANÇA, `super` E ALTO ACOPLAMENTO

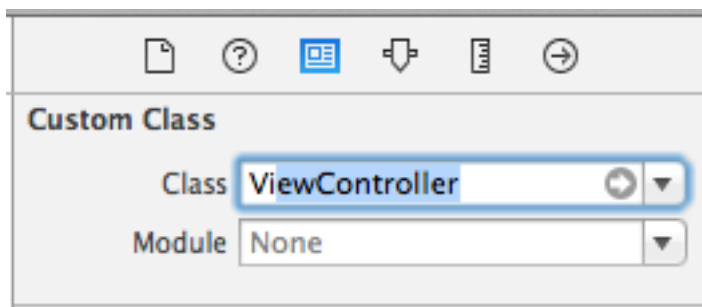
A palavra-chave `override` indica que estamos sobrescrevendo um método que já existia em nossa classe mãe.

O que é esse tal de `super`? Ele invoca o método com a assinatura (nome e parâmetros) definido na classe mãe (ou pai, tanto faz), no nosso caso `UITableViewController`. Como não sabemos o que o método faz na classe mãe, é importante o chamarmos para que ele inicialize o que for necessário.

A palavra `super` entrega ao programador atento a existência um alto acoplamento entre classes que usam herança: a classe filha tem que entender muito bem o código da classe mãe para saber quando ela tem que chamar `super` obrigatoriamente e quando ela não pode chamar `super`. O uso indiscriminado de herança pode levar a problemas de manutenção de código a longo prazo.

<http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-heranca/>

Uma vez salvas as alterações no `ViewController`, de volta ao storyboard, a IDE sugere o `ViewController` como classe que customiza o comportamento de nosso `controller`:



Rodamos o programa e temos nossa tela sem nenhuma célula, mas já co-

nectada ao `controller`, podemos conferir isso analisando o log, que mostra nossa mensagem impressa durante o `viewDidLoad`:



Precisamos agora do `Array` que representará os alimentos a serem mostrados. Como esse projeto novo não possui nossas refeições, começaremos com uma simples, de `Strings`:

```
let meals = [ "eggplant brownie", "zucchini muffin"]
```

Mas como armazenar essas refeições de tal maneira que elas fiquem na memória enquanto nosso `controller` existir? Podemos colocar nosso modelo que está sendo exibido (neste caso, a `Array` de `Strings`) como propriedades de nossa classe:

```
class ViewController: UITableViewController {  
  
    let meals = [ "eggplant brownie", "zucchini muffin"]  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
        println("view did load")  
    }  
}
```

Assim como fizemos com a tabela com tamanho fixo, queremos dizer a ela quantas linhas desejamos ter. Mas em vez de notificarmos a tabela, nós

seremos perguntados por ela, isto é, escrevemos uma função que devolve o número de linhas que nossa `Table View` terá:

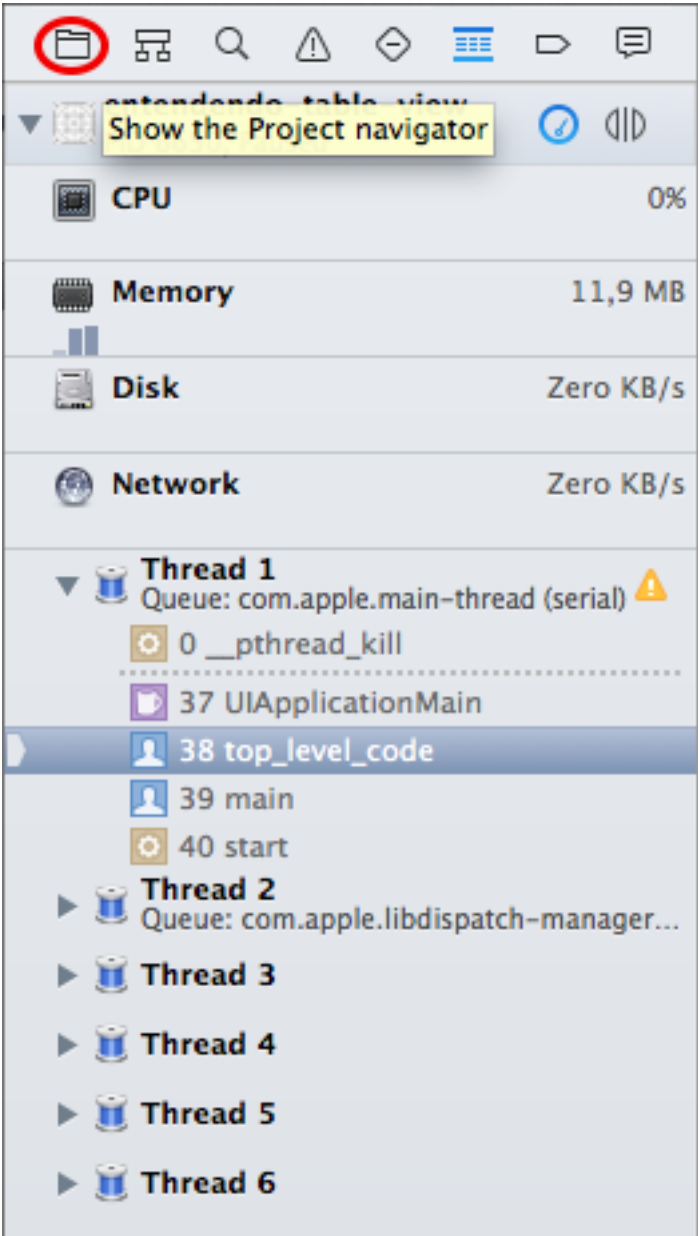
```
func howManyLinesDoWeHaveInOurTable() -> Int {  
    return meals.count  
}
```

Esse método já existe com o retorno padrão `0`, e desejamos reescrevê-lo. Seu nome é `tableView` e ele recebe dois parâmetros que não utilizaremos:

```
override func tableView(tableView: UITableView,  
    numberOfRowsInSection section: Int) -> Int {  
    return meals.count  
}
```

Ao tentarmos rodar, temos um problema. Acontece que falamos o número de células desejadas mas não o valor de cada uma delas. Aí não dá... Vamos definir o valor de cada uma de maneira análoga ao que fizemos com uma tabela de tamanho.

Podemos parar o programa clicando no `stop` e voltar a ver o nosso projeto clicando em `Project Navigator` (primeiro ícone da barra de navegação à esquerda).



Reescreveremos um outro método da `tableView`. Em vez dos dois parâmetros anteriores (tabela e número de linhas), recebemos a tabela e a linha da qual ele quer saber a célula a ser utilizada:

```
override func tableView(tableView: UITableView,
                        cellForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCell {
    let row = indexPath.row
    let meal = meals[ row ]
}
```

Agora que extraímos a linha em que estamos interessados, criamos uma nova `view` do tipo `UITableViewCell` com o estilo padrão e sem passar um identificador:

```
override func tableView(tableView: UITableView,
                        cellForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCell {
    let row = indexPath.row
    let meal = meals[ row ]

    var cell = UITableViewCell(
        style: UITableViewCellStyle.Default,
        reuseIdentifier: nil)
}
```

Já temos a célula. Vamos alterar seu texto para o valor da refeição e retornamos:

```
override func tableView(tableView: UITableView,
                        cellForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCell {
    let row = indexPath.row
    let meal = meals[ row ]

    var cell = UITableViewCell(
        style: UITableViewCellStyle.Default,
        reuseIdentifier: nil)
    cell.textLabel.text = meal
```

```
        return cell
    }
}
```

O código final de nosso `controller` é:

```
import UIKit

class ViewController: UITableViewController {

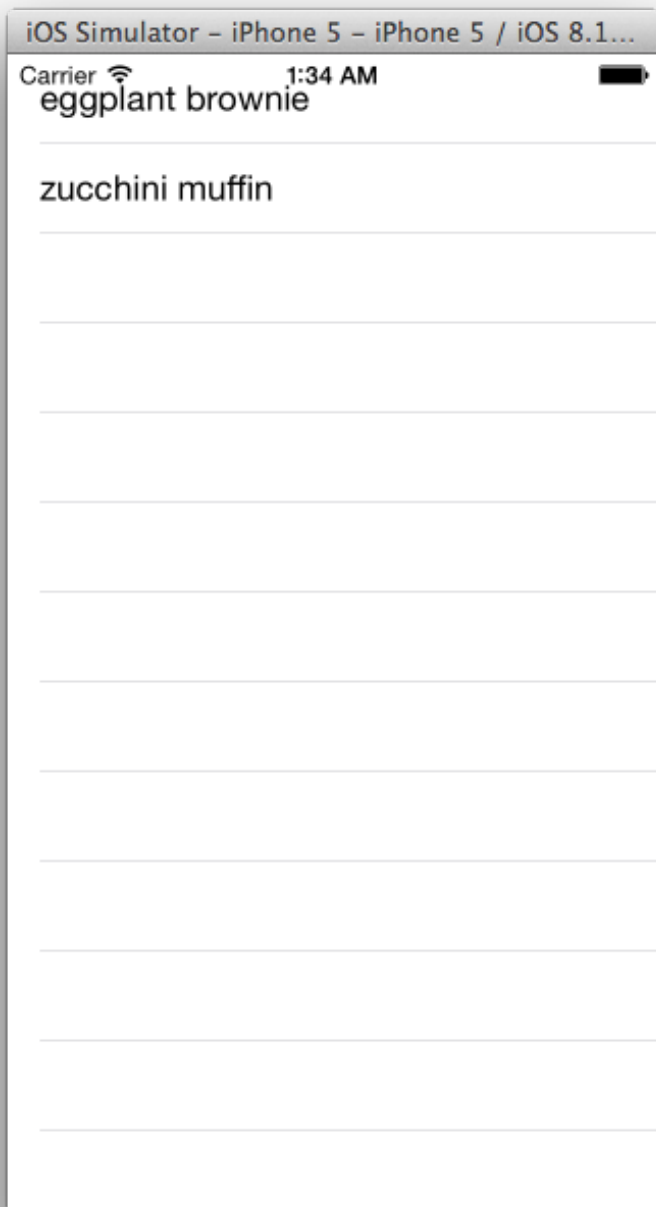
    let meals = [ "eggplant brownie", "zucchini muffin" ]

    override func viewDidLoad() {
        super.viewDidLoad()
        println("view did load")
    }

    override func tableView(tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int {
        return meals.count
    }

    override func tableView(tableView: UITableView,
        cellForRowAtIndexPath indexPath: NSIndexPath)
        -> UITableViewCell {
        let row = indexPath.row
        let meal = meals[ row ]
        var cell = UITableViewCell(
            style: UITableViewCellStyle.Default,
            reuseIdentifier: nil)
        cell.textLabel.text = meal
        return cell
    }
}
```

Rodamos novamente e temos uma tabela dinâmica, com os dois elementos que representam as duas refeições!



VIEW

Uma view é um componente visual qualquer. Tanto o `controller` é uma view quanto a `Label` e uma célula são views. Views podem ser colocadas umas dentro das outras para compor uma view mais complexa. Nesse nosso caso, temos um `controller` com uma `table view` com diversas `table view cell`. Enquanto criamos o `UITableViewController` visualmente, arrastando-o da barra de componentes, criamos as `UITableViewCell` programaticamente.

Agora é hora de aplicar isso tudo em nosso projeto!

6.2 RESUMO

Vimos como criar uma tabela de tamanho fixo, uma tarefa que pode ser aplicada quando temos determinadas características a serem configuradas ou opções a serem selecionadas dentro de um conjunto predeterminado, cujo desenvolvedor pode escrever em tempo de desenvolvimento.

Passamos pela criação de um `controller` que já possui um único elemento, a tabela em si, o `UITableViewController` e aprendemos a indicar qual é o ponto de início de nossa aplicação.

Criamos um `controller` que mostrava uma tabela refletindo o número de elementos contidos em um array, assim como customizamos o conteúdo da célula dessas tabelas de acordo com o conteúdo desse array.

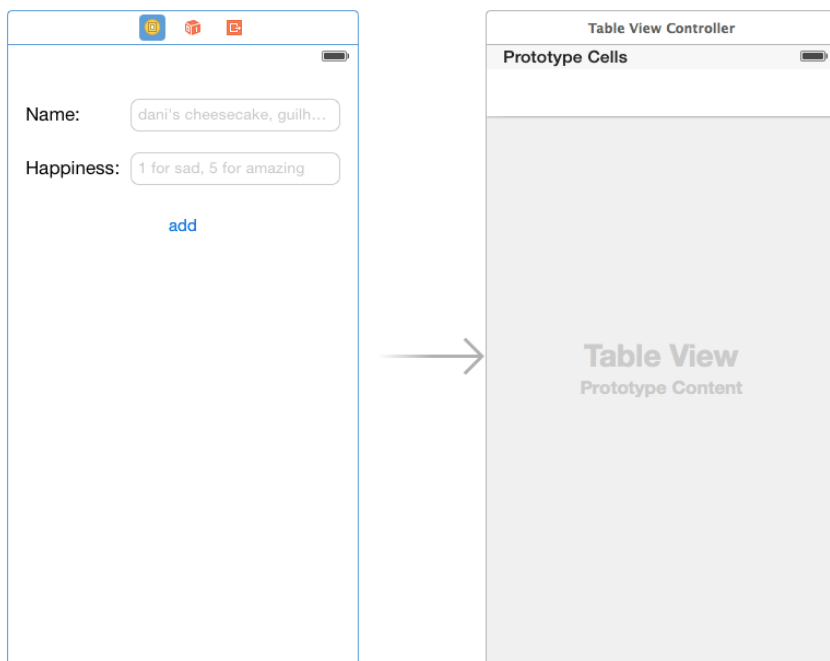
CAPÍTULO 7

Projeto: lista de refeições

Agora que já somos capazes de criar uma nova refeição, e sabemos criar uma tabela, desejamos criar uma visualização capaz de mostrar todas as refeições.

Como fazer isso? Se desejamos uma nova visualização no nosso projeto original (`eggplant-brownie`), podemos adicionar um novo `ViewController`, um que inclua dentro dele um componente que seja uma listagem de refeições, uma tabela. Podemos simplesmente adicionar um `TableViewController`. Abrimos nosso `Main.storyboard`, procuramos o `TableViewController` na lista de componentes no campo inferior direito e o puxamos para nosso `storyboard`. Lembre-se de já mudar o `size` dele para o de `iPhone` que estamos utilizando até agora.

Diferentemente do que acontecia até agora nessa aplicação, desejamos que ela comece com essa view, e não a anterior, a view de adicionar refeição, portanto movemos a seta de *starting point* de nossa aplicação para esse novo `controller`.

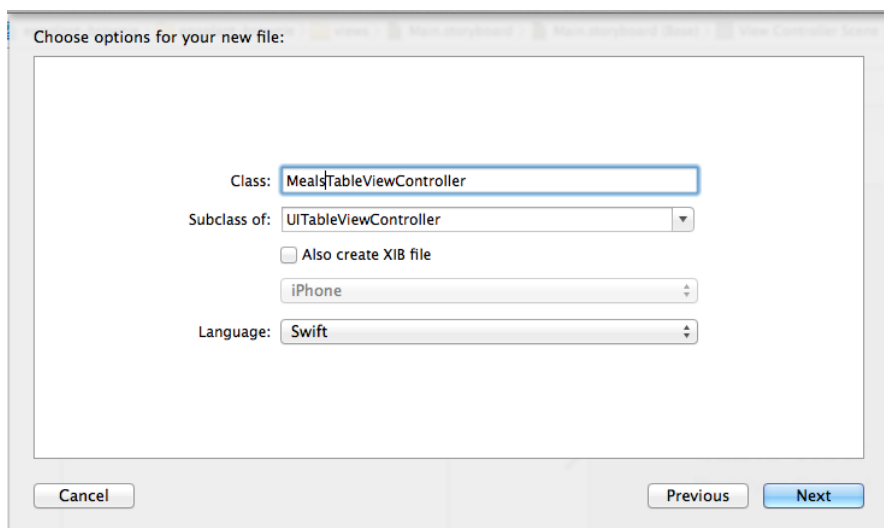


Queremos que essa tabela seja dinâmica, e à medida que adicionemos novos elementos na lista de refeições, ela reflita essa nossa lista. Temos diversos passos para implementar:

- Criar o código de nosso `controller`;
- Criar um array de refeições;
- Fazer a tabela refletir os dados do array de refeições;
- Permitir clicar em um botão e ir para a tela de adicionar refeição;
- Ao voltar da tela de adicionar refeição, atualizar o array de refeições.

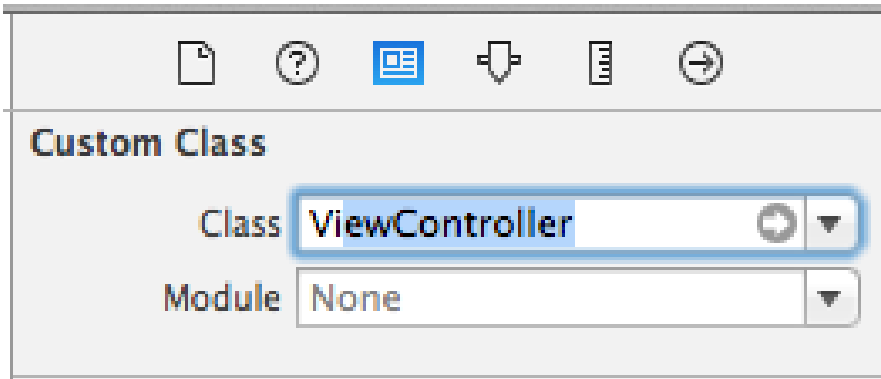
Começamos criando o código de nosso `controller`. Na pasta `viewcontrollers`, clicando com o botão direito ou com um *control-click*, escolhemos `New File....` Selecionamos `Source` dentro de

iOS e, clicando duas vezes em `Cocoa Touch Class`, damos o nome `Meals` e escolhemos o tipo `UITableViewController`, resultando em `MealsTableViewController`. Confirmamos que escolhemos `Swift` como linguagem:



O Xcode cria um novo `ViewController` que agora herda o comportamento de `UITableViewController`. A classe criada possui diversos métodos implementados e comentados, mas como, em geral, não vamos usá-los, podemos apagar todos os métodos. Já já escreveremos os métodos que nos interessam.

Precisamos conectar a view do nosso `storyboard` com o `controller` que acabamos de criar, fazendo isso na propriedade de classe customizada, como fizemos anteriormente no `storyboard`:



Cuidado para não tentar customizar a classe somente de sua `UITableView`, o Xcode não dará a sugestão que deseja. Escolha seu `MealsTableViewController` clicando no ícone amarelo que representa selecionar o view controller para só então alterar a classe que a identifica.

Se rodarmos a aplicação, percebemos que não aparece nenhuma linha preenchida. Agora é a hora de criarmos nossas refeições iniciais:

```
class MealsTableViewController: UITableViewController {  
  
    var meals = [ Meal(name: "Eggplant brownie", happiness: 5),  
                  Meal(name: "Zucchini Muffin", happiness: 3)]  
  
}
```

Implementamos também o código para definir o número de linhas refletindo o total de refeições em nosso array:

```
override func tableView(tableView: UITableView,  
                        numberOfRowsInSection section: Int) -> Int {  
    return meals.count  
}
```

O método que devolve o conteúdo de cada linha extrairá o nome de cada refeição:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCell {
    let row = indexPath.row
    let meal = meals[ row ]

    var cell = UITableViewCell(
        style: UITableViewCellStyle.Default,
        reuseIdentifier: nil)
    cell.textLabel.text = meal.name
    return cell
}
```

Rodamos a aplicação: agora temos uma tela com todas as refeições! Mas ainda temos algo de estranho por aqui, como faremos para adicionar uma nova refeição? Precisamos de uma barra de navegação, que veremos em breve.

7.1 RESUMO

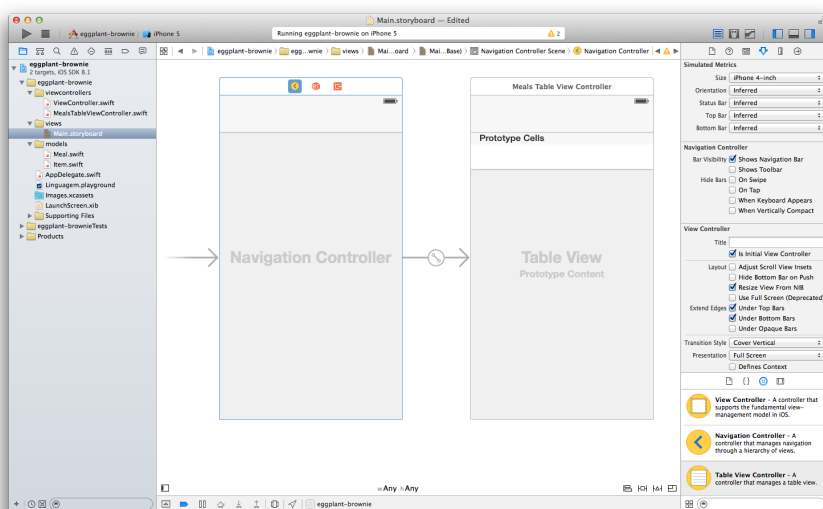
Vimos neste capítulo como criar um `TableViewController` que mostra todas as refeições, além de configurá-lo para ser o novo ponto de entrada de nossa aplicação. Sobrescrevendo métodos de nosso `TableViewController`, fomos capazes de configurar a quantidade de linhas e o conteúdo de cada célula de nossa tabela para seguir os dados presentes em um array de objetos.

CAPÍTULO 8

Navegando entre telas

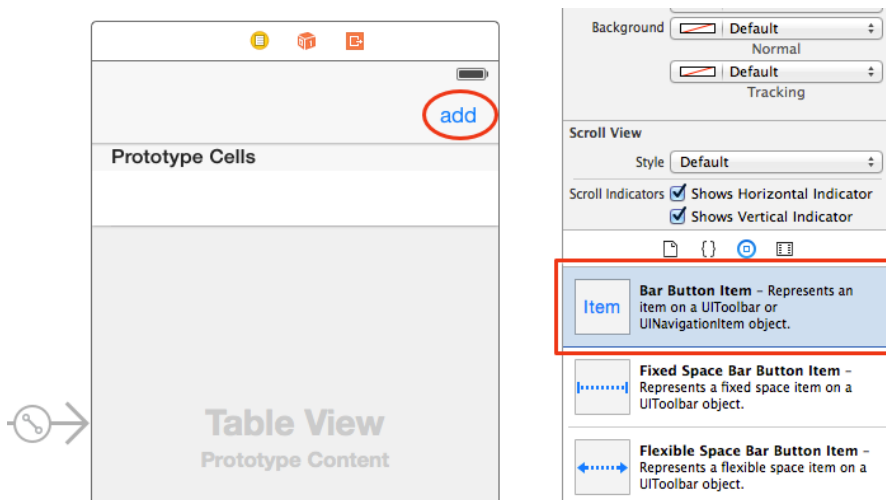
Precisamos fornecer uma maneira para nosso usuário sair da listagem de refeições e ir para a tela de nova refeição: queremos trabalhar com a navegação entre telas. Ele precisa navegar entre a tela de listagem e a de nova refeição, tanto a ida quanto a volta. Queremos uma barra de navegação, e isso pode ser feito através de um `Navigation Controller`.

Começamos clicando em nossa tabela e, no menu `Editor`, submenu `Embed In`, escolhemos `Navigation Controller`, fazendo com que nosso `controller` tenha agora uma barra de navegação. Como de costume, mude o tamanho do seu `Navigation Controller`. Note também que o Xcode já moveu o ponto de entrada do programa para a tela com navegação:

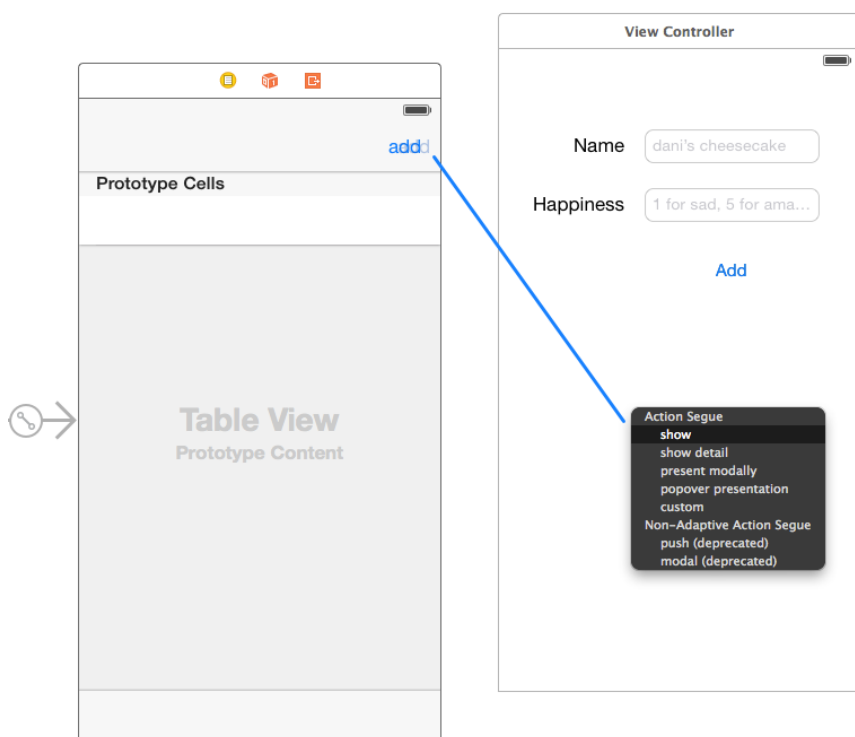


Primeiro, foi criado um `Navigation Controller` e nosso `MealsViewController` foi alterado para ser a raiz (`root`), o ponto de entrada desse `Navigation Controller`. Na prática, o `Navigation Controller` é só uma capa que veste o nosso controller inicial. A partir daí, a navegação é feita sempre mantendo essa capa de navegação. Portanto, não faremos nada agora no `Navigation Controller` em si, mas no nosso controller antigo.

Note que, no nosso controller antigo, temos uma barra de navegação, na qual podemos adicionar um componente chamado `Bar Button Item`, e mudamos seu nome para `Add`.



Agora, com a tecla `Control` apertada, arrastamos o botão para nossa tela de `Adicionar Refeição`, escolhendo a opção `Action Segue -> show`, isto é, ao clicar no botão, mostre essa tela.



Podemos perceber que alguns campos do formulário ficaram atrás da barra de navegação. Para fazer com que apareçam embaixo da barra, clicamos no View Controller do formulário e, no menu de propriedades (Attributes Inspector), desmarcamos Under Top Bars. Pronto, agora os campos são deslocados automaticamente para baixo.

Rodamos o programa e percebemos que, ao clicar no botão, vamos direto para a tela de nova refeição. Perfeito, mas ao clicar em adicionar, não voltamos para a tela anterior. Temos que fazer a volta.

8.1 NAVEGANDO COM UMA PILHA DE TELAS

Para fazer a volta, devemos dizer para nosso `controller` de navegação que desejamos voltar atrás na tela atual. Isto é, cada vez que mostramos uma tela de maneira tradicional (chamada de `push`), estamos empilhando uma tela em cima da outra.

Quando desejamos remover a tela atual de cima da anterior, usamos o método `pop`. Portanto, em nossa função `add`, no `ViewController`, devemos pegar nosso `navigationController` (que é opcional) e falar para ele remover o `view controller` de maneira animada:

```
self.navigationController!.popViewControllerAnimated(true)
```

Cuidado com métodos de nome parecido com o `popViewControllerAnimated`, a escolha de outro método trará um outro efeito, claro.

Mas, opa, nosso atual `controller` pode e pode não ter um `navigationController`, sendo assim estamos acessando uma variável que pode destruir nossa aplicação. Podemos nos proteger usando a construção `let`:

```
if let navigation = self.navigationController {  
    navigation.popViewControllerAnimated(true)  
}
```

Agora sim, temos nosso método `add` que extrai os dados, transforma o número e volta pra tela anterior, sempre com a garantia de que erros de inicialização ou de entrada do usuário não quebrarão nossa app:

```
@IBAction func add() {  
    if nameField == nil || happinessField == nil {  
        return  
    }  
  
    let name = nameField!.text  
    let happiness = happinessField!.text.toInt()  
    if happiness == nil {  
        return  
    }  
}
```

```
}

let meal = Meal(name: name, happiness: happiness!)
println("eaten: \(meal.name) \(meal.happiness)")

if let navigation = self.navigationController {
    navigation.popViewControllerAnimated(true)
}
}
```

NAVIGATION OPCIONAL?

E a pergunta de 1000 reais: como assim o `navigationController` é uma variável opcional do meu `controller`? Acontece que, devido à decisão da API de fazer com que um `UIViewController` **sempre** tenha essa variável, ela pode estar setada ou não... pois você pode estar dentro de um `UINavigationController` ou não. Em tempo de compilação, a API não sabe disso, portanto deixa a variável como opcional.

A API poderia ser recriada (quebrando compatibilidade com versões anteriores, o que é muitas vezes indesejado) para que a variável só existisse em compilação caso seu `controller` estivesse realmente dentro de um `navigation controller`, evitando a variável opcional e possíveis erros em tempo de execução.

Mas como toda decisão de quebrar compatibilidade é perigosa, por enquanto convivemos com tal variável opcional e o perigo de acessá-la sem verificar seu valor.

Testamos novamente nossa aplicação e, agora, quando clicamos no botão de adicionar, ele desempilha nossa tela, mostrando a tela de refeições novamente.

8.2 RESUMO

Vimos como podemos criar um `NavigationController` que permite a navegação entre diversos `View controllers`. Apresentamos um controller com o `push` de um segue e voltamos para a tela anterior com o `pop` programático.

CAPÍTULO 9

Design pattern: delegate

Nosso próximo passo é conseguir atualizar nossa tela com a refeição que acaba de ser criada. Como fazer isso? Pode ser de diversas maneiras: a primeira é fazer com que a lista fique perguntando a todo o momento se uma nova refeição foi criada. Esta técnica é chamada de `polling` e parece não se encaixar de jeito nenhum aqui. Uma segunda forma é pedir para que o formulário avise que uma nova refeição foi criada, e quem quiser pode escutar este evento e fazer as ações necessárias.

Esta última é a implementação do padrão chamado `Delegate`: o formulário delega a responsabilidade de adicionar o resultado no array para alguém que sabe terminar esse trabalho, atualizando a tela com a nova refeição. Esse *design pattern* lembra o `Observer` do livro “Design Patterns: Elements of Reusable Objected-Oriented Software”, onde um outro objeto é notificado de um evento que ocorreu.

Primeiro, precisamos que nosso `MealsTableViewController` seja

capaz de adicionar uma nova refeição ao array existente. Adicionamos nele um método `add`:

```
func add(meal: Meal) {  
    meals.append(meal)  
}
```

Agora, o formulário do `ViewController`, após adicionar a nova refeição, deve invocar uma ação que faça o registro dela para quem precisar ser informado deste evento, ou seja, para o `delegate` desta ação:

```
@IBAction func add() {  
    if nameField == nil || happinessField == nil {  
        return  
    }  
  
    let name = nameField!.text  
    let happiness = happinessField!.text.toInt()  
    if happiness == nil {  
        return  
    }  
  
    let meal = Meal(name: name, happiness: happiness!)  
    println("eaten: \(meal.name) \(meal.happiness)")  
  
    delegate.add(meal)  
  
    if let navigation = self.navigationController {  
        navigation.popViewControllerAnimated(true)  
    }  
}
```

Mas onde está a variável `delegate`? Precisamos criá-la como propriedade, para que alguém possa configurá-la:

```
var delegate:MealsTableViewController  
  
@IBAction func add() {  
    if nameField == nil || happinessField == nil {
```

```
        return
    }

    let name = nameField!.text
    let happiness = happinessField!.text.toInt()
    if happiness == nil {
        return
    }

    let meal = Meal(name: name, happiness: happiness!)
    println("eaten: \(meal.name) \(meal.happiness)")

    delegate.add(meal)

    if let navigation = self.navigationController {
        navigation.popViewControllerAnimated(true)
    }
}
```

Mas se nosso `delegate` só vai ser configurado após o `ViewController` ser criado, precisamos deixá-lo como variável opcional:

```
var delegate:MealsTableViewController?

@IBAction func add() {
    if nameField == nil || happinessField == nil {
        return
    }

    let name = nameField!.text
    let happiness = happinessField!.text.toInt()
    if happiness == nil {
        return
    }

    let meal = Meal(name: name, happiness: happiness!)
    println("eaten: \(meal.name) \(meal.happiness)")
}
```

```
        delegate.add(meal)

        if let navigation = self.navigationController {
            navigation.popViewControllerAnimated(true)
        }
    }
}
```

E já sabemos que, se é opcional, temos que cuidar com carinho:

```
var delegate:MealsTableViewController?

@IBAction func add() {
    if nameField == nil || happinessField == nil {
        return
    }

    let name = nameField!.text
    let happiness = happinessField!.text.toInt()
    if happiness == nil {
        return
    }

    let meal = Meal(name: name, happiness: happiness!)
    println("eaten: \(meal.name) \(meal.happiness)")

    if delegate == nil {
        return
    }

    delegate!.add(meal)

    if let navigation = self.navigationController {
        navigation.popViewControllerAnimated(true)
    }
}
```

Rodamos o nosso programa, a mensagem de log aparece mas não saímos da tela. Clicamos no botão ‘Back’ e não temos o elemento na tabela. Isso acontece porque a nossa variável opcional, o `delegate`, não foi configurada!

Portanto o `'println'` foi executado e logo depois a execução parou a chamada a nossa função `'add'` pois `'delegate == nil'` era verdadeiro.

Quem será então nosso `delegate`? Quem observará a view de adicionar refeição para ser notificado quando uma nova refeição foi criada? Nosso `MealsTableViewController`.

9.1 CONFIGURANDO UM DELEGATE VIA SEGUE

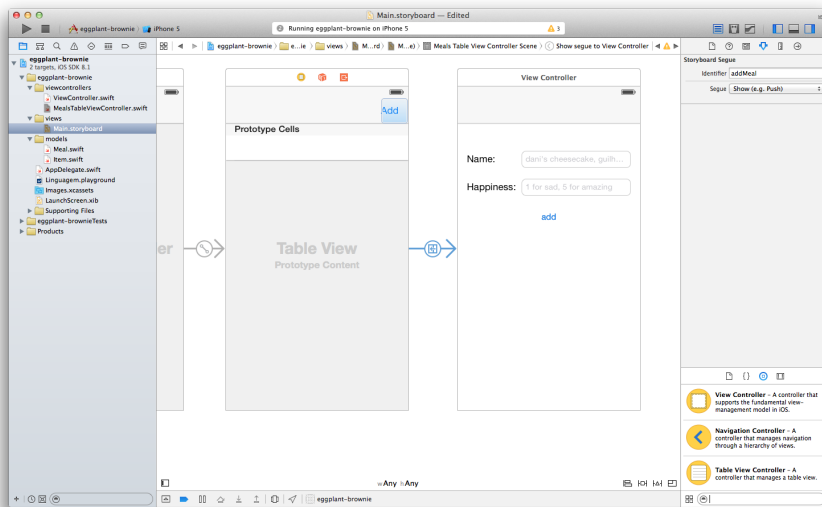
Precisamos agora notificar o `ViewController` de que nós, `MealsTableViewController`, somos o seu *Observer*, o seu `delegate`. Mas como fazer isso se a conexão entre os dois `controllers` é feita via uma seta (um `segue`) no storyboard, e não via programação? Como passar um parâmetro via programação para outro `controller`, sendo que a conexão é feita via arrasta e solta?

Se tivéssemos acesso ao `controller` de destino na hora da navegação, poderíamos pegar o objeto e colocar nosso `delegate` nele.

Nossos `controllers` nos disponibilizam um momento de alegria antes de redirecionarem para outra view. Ao se preparar para seguir um `segue`, o `controller` invoca um método chamado `prepareForSegue`, onde podemos sobrescrever e fazer algo com o que será mostrado.

```
override func prepareForSegue(segue:
    UIStoryboardSegue, sender: AnyObject?) {
}
```

Repare que a função recebe um `segue`, aquele que ele está seguindo. Mas como sabemos qual ele está seguindo? Precisamos de algum identificador, de um `id`. Vamos em nosso storyboard, clicamos no `segue` e, em seguida, no menu `Attributes Inspector`, vamos configurar seu identificador como `addMeal`.



Agora sim, em nosso `prepareForSegue` do `MealsTableViewController` podemos condicionar a execução ao segue que desejamos:

```
override func prepareForSegue(segue: UIStoryboardSegue,
    sender: AnyObject?) {
    if(segue.identifier == "addMeal") {
    }
}
```

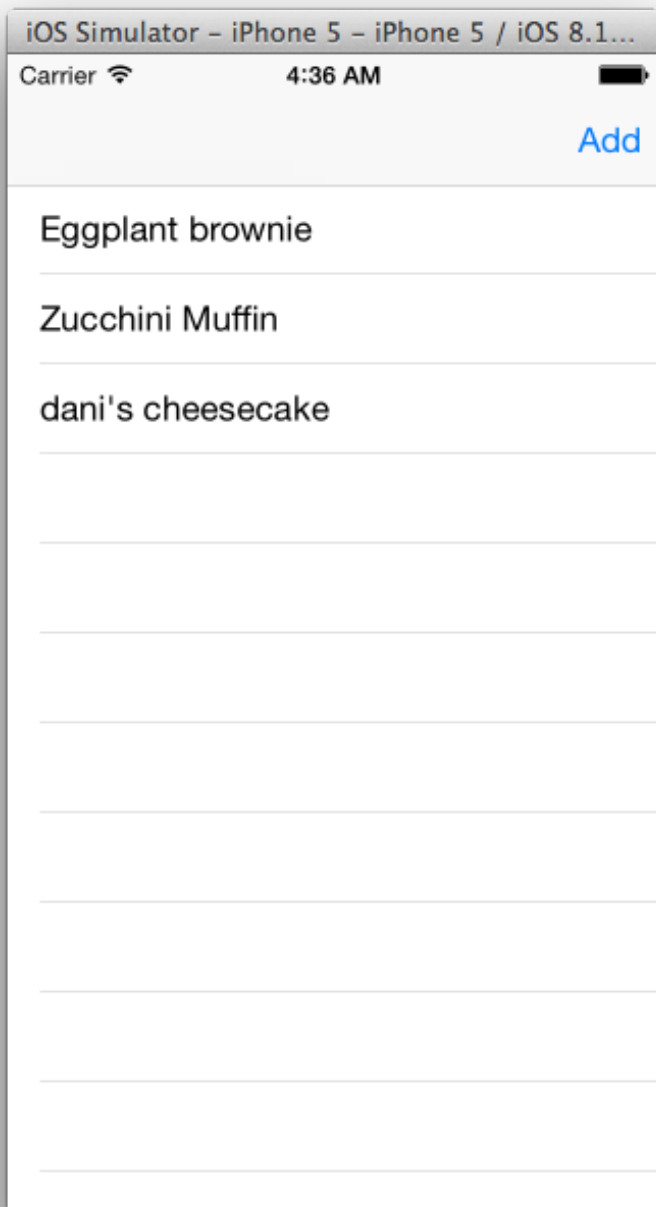
O que falta agora é pegar nosso `controller` de destino, que podemos obter da própria `segue`:

```
override func prepareForSegue(segue: UIStoryboardSegue,
    sender: AnyObject?) {
    if(segue.identifier == "addMeal") {
    let view = segue.destinationViewController as ViewController
    }
}
```

E setarmos nosso delegate:

```
override func prepareForSegue(segue: UIStoryboardSegue,  
    sender: AnyObject?) {  
    if(segue.identifier == "addMeal") {  
let view = segue.destinationViewController as ViewController  
        view.delegate = self  
    }  
}
```

Pronto. Testamos nossa aplicação, a refeição é adicionada com sucesso:



BOA PRÁTICA: RECARREGUE OS DADOS DE UMA TABELA

Nosso código de adicionar nova refeição está bem simples:

```
func add(meal: Meal) {  
    meals.append(meal)  
}
```

Mas a tabela não é redesenhada automaticamente só por ter adicionado um elemento novo em um array. No nosso caso, ela é redesenhada pois retiramos um `ViewController` que estava na frente, forçando a aparição da tabela novamente. Como o método `add` pode ser chamado em situações em que o `pop` não seria chamado, é interessante garantir que, toda vez que adicionamos algo através desse método, a tabela seja redesenhada. Para isso, invocamos o método `reloadData` de nossa `tableView`:

```
func add(meal: Meal) {  
    meals.append(meal)  
    tableView.reloadData()  
}
```

9.2 CODE SMELL: NOMES GENÉRICOS DEMAIS E COMO EXTRAIR UM PROTOCOLO

Mas qual o tipo da variável `delegate`? Sério mesmo que é `MealsTableViewCellController`? Isto é, no código de um `controller` temos uma referência para o outro, e no outro para o um, explicitamente. Quanto maior o acoplamento entre duas classes, mais complexo fica mudá-las sem uma alterar a outra, e isso é algo que queremos evitar, claro.

Esta variável precisa ser de um tipo que possua obrigatoriamente o método `add` para que possamos invocá-lo, mas não precisa ter todo o resto que um `MealsTableViewCellController` tem. Para podermos garantir que o nosso `delegate` terá com toda a certeza o método `add`, temos que fa-

zer com que ele assine uma espécie de um contrato que o obrigue a ter este método. Um contrato funciona como uma interface: todos que a implementarem devem cumprir com os métodos definidos lá dentro. Este contrato é chamado de `Protocol`.

Precisamos extrair o protocolo que tenha o método `add` que recebe um `Meal`:

```
protocol MyDelegate {  
    func add(meal: Meal)  
}
```

Como este `protocol` será utilizado para que nosso `ViewController` consiga invocar o `delegate`, vamos declará-lo no mesmo arquivo de nosso `ViewController`, o `ViewController.swift` logo antes da declaração da classe:

```
protocol MyDelegate {  
    func add(meal: Meal)  
}  
  
class ViewController: UIViewController {  
    //restante do código  
}
```

Falta decidir qual nome usar para nossa interface, nosso `protocol`, `MyDelegate` não parece um nome que diz muito sobre nosso domínio ou o que o protocolo faz. Existe uma convenção da Apple para nomes de protocolos, criada na época do Objective-C: <http://bit.ly/swiftcodenamingbasics>

Mas para os casos de `Delegate` a regra que a Apple utiliza em sua API, é de colocar o nome da classe que terá seu comportamento delegado mais a palavra `Delegate`. Veremos em breve o `UITableViewDelegate`, por exemplo. Em nosso caso, temos então `ViewControllerDelegate`:

```
protocol ViewControllerDelegate {  
    func add(meal: Meal)  
}
```

Nomenclatura de protocolos

O problema dessa convenção ao mesmo tempo oficial e não oficial no padrão do `delegate` está ligado à sua desconexão do nosso domínio. O que um `ViewControllerDelegate` faz? Não tenho ideia. O que um `AddAMealDelegate` faz? Ele adiciona uma refeição. A desconexão do nosso projeto com o domínio de negócios é um dos fatores que dificulta a manutenção do código. Além disso, uma das maneiras de ver esse `delegate` é como um `Observer`; notifique-me quando uma nova refeição for criada. Teríamos um protocolo totalmente diferente:

```
protocol MealsObserver {  
    func created(meal: Meal)  
}
```

O resultado seria o mesmo, sendo uma mera discussão de “qual *design pattern* estou usando”, portanto de “qual nome devo usar”.

No nosso caso, vamos fugir do padrão `ViewControllerDelegate` por um único motivo: ele nos induz a manter um único `delegate`. E se desejássemos ter dois `delegates` para uma única tela, como por exemplo um dashboard de funções de nossa aplicação, em que cada gráfico clicado gera uma ação totalmente diferente? Usando o padrão mencionado, teríamos um único `Delegate`? O desenvolvedor, para tentar seguir o padrão, se sentiria tentado a manter o código todo em um único lugar.

Em vez de colocarmos muita responsabilidade em um único lugar, vamos lutar pela separação de responsabilidades e por protocolos menores, sendo assim, já escolheremos um nome de domínio para nosso `Delegate`.

Queremos um nome de `Delegate` que faça sentido para nosso domínio, um protocolo que define quem será capaz de adicionar refeições, um `AddAMealDelegate`. Nosso protocolo é, na verdade:

```
protocol AddAMealDelegate {  
    func add(meal: Meal)  
}
```

Agora podemos colocar o tipo de nossa variável:

```
var delegate: AddAMealDelegate?
```

Temos também que falar que nossa classe `MealsTableViewController` adota esse protocolo:

```
class MealsTableViewController: UITableViewController,
    AddAMealDelegate {

    // ...
}
```

Portanto, o código de nosso `ViewController` fica:

```
import UIKit

protocol AddAMealDelegate {
    func add(meal: Meal)
}

class ViewController: UIViewController {

    @IBOutlet var nameField: UITextField!
    @IBOutlet var happinessField: UITextField!

    var delegate:AddAMealDelegate?

    @IBAction func add() {
        if nameField == nil || happinessField == nil {
            return
        }

        let name = nameField!.text
        let happiness = happinessField!.text.toInt()
        if happiness == nil {
            return
        }

        let meal = Meal(name: name, happiness: happiness!)
        println("eaten: \(meal.name) \(meal.happiness)")

        if delegate == nil {
            return
        }
    }
}
```

```
    }

    delegate!.add(meal)

    if let navigation = self.navigationController {
        navigation.popViewControllerAnimated(true)
    }
}
}
```

Testamos novamente nossa aplicação e o código está funcionando.

MÁ PRÁTICA: MUITOS SEGUES COM PARÂMETROS

O uso de muitos `segues` gera um código cheio de `ifs`, também conhecido como `switch`. Ambos são indicadores de que existe muita responsabilidade em um único ponto de nosso código, o que o torna cada vez mais difícil de manter. Existem diversas técnicas para evitar esses `ifs`, como o uso de polimorfismo em Orientação a Objetos. Veremos mais à frente como evitar a passagem de parâmetro via `prepareForSegue` fazendo como os desenvolvedores sêniores, usando programação normal.

9.3 RESUMO

Conhecemos aqui o padrão `observer` que é aplicado com o nome de `Delegate` para notificar uma outra tela de uma ação realizada na tela atual. Quem observa a notificação não precisava necessariamente ser outra tela, bastava implementar um `protocol` que definimos.

Precisamos dar uma identificação para o `segue` para que ele possa ser utilizado durante a preparação de redirecionamento, o momento no qual configuramos em nosso *view controller* quem será o `delegate` a ser invocado.

O `delegate` é o coração da comunicação entre telas, sem ele seríamos obrigados a apelar para variáveis globais - a maior de todas as quebras de encapsulamento.

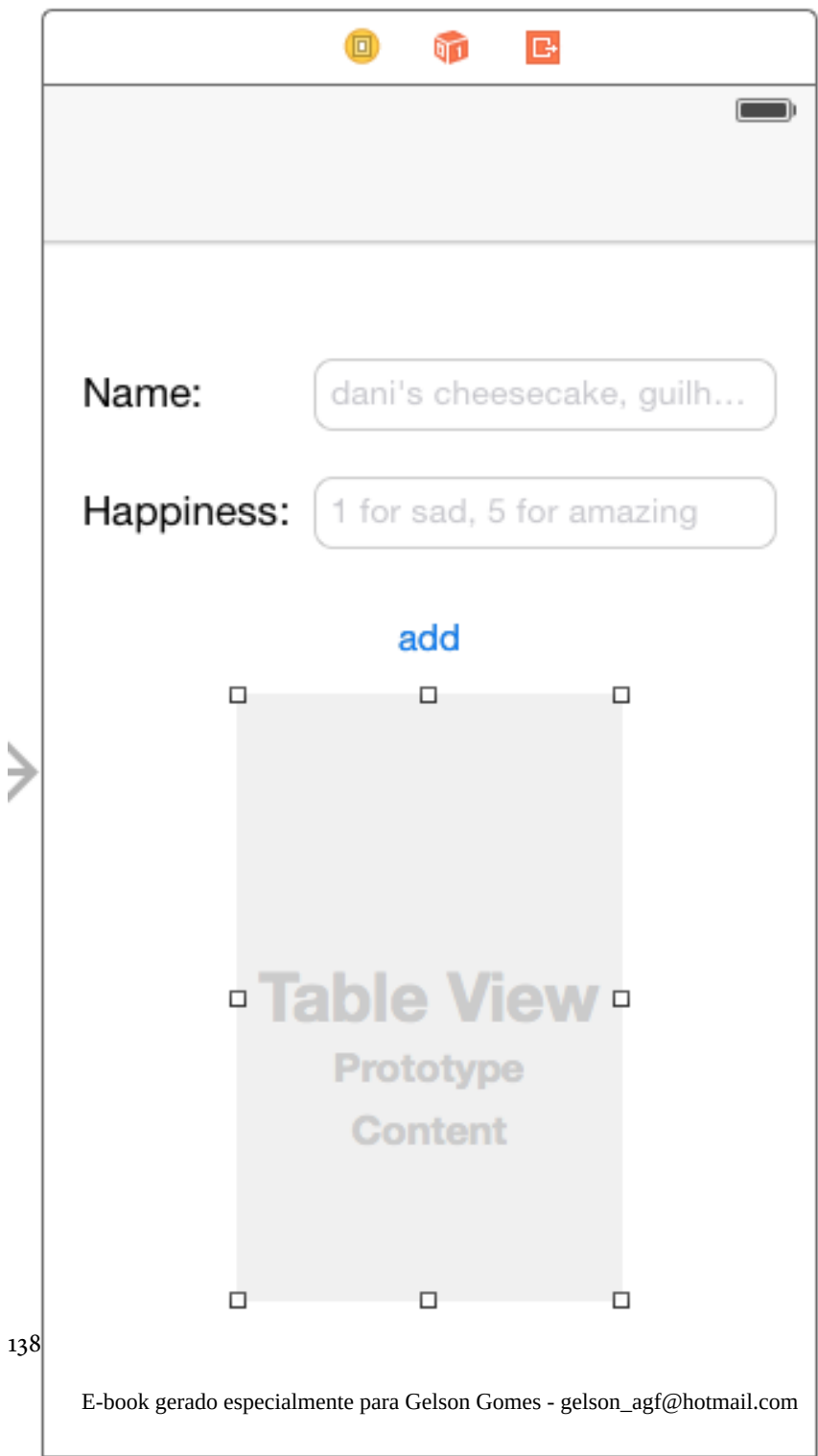
O post da Caelum <http://bit.ly/singletons-static> tem uma explicação sobre alguns motivos pelos quais variáveis globais são perigosas para a manutenção de uma aplicação.

Os nomes de classes, métodos e variáveis são importantes para a manutenção de um código em longo prazo e a discussão é uma que ainda está em aberto. Por um lado, temos uma convenção que nos leva a acumular responsabilidades em um único protocolo, e posteriormente em uma única classe, por outro lado, um nome que foge da convenção.

CAPÍTULO 10

Relacionamento um para muitos: lista de alimentos

Ao adicionar uma nova refeição, devemos escolher quais são os alimentos que a compõem. Isto é, desejamos mostrar a lista de itens na tela de nova refeição. Se vamos mostrar uma lista, adicionaremos uma tabela. Mas como fazer isso se o `controller` que vimos até agora para a criação de tabelas permitia somente a existência de tabelas? Vamos adicionar um único componente novo ao `controller` de nova refeição: uma `TableView`, arrastando, como sempre, a view do canto inferior direito até nosso `view controller` no storyboard:

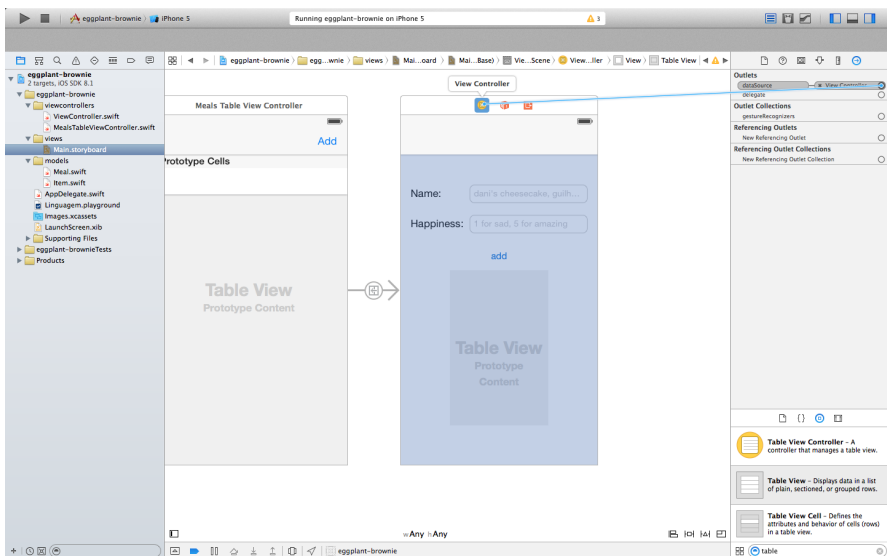


Como toda `UITableView` dinâmica, devemos indicar quem será o objeto responsável por implementar os métodos que descrevem uma tabela, como o número de células e seu conteúdo.

Como queremos que nosso *view controller* seja responsável por tais métodos, devemos primeiro fazer com que ele implemente os protocolos adequados ou herde de `UITableViewController`. Mas não somos bem um `UITableViewController`, certo? Não queremos assumir mais responsabilidades do que realmente devemos, portanto preferimos implementar somente o protocolo que fornece os dados da tabela, a fonte de dados, o `data source`, o protocolo chamado `UITableViewDataSource`:

```
class ViewController: UIViewController, UITableViewDataSource {  
    // ...  
}
```

No storyboard, selecionamos nosso `TableView` e, na aba Connections Inspector (último ícone na barra de propriedades), puxamos o `dataSource` para o ícone de nosso View Controller (aquele amarelo no topo da tela de adicionar):



Precisamos implementar agora os métodos `tableView` que retornam a quantidade de alimentos dentre os quais o usuário selecionará os que entram na refeição, além do método que devolve o conteúdo de cada célula. Para isso, vamos primeiro criar um array com todos os alimentos possíveis. Segundo minha esposa confeitadeira, o brownie de berinjela pode combinar com uma cobertura de chocolate, enquanto o muffin de abobrinha vai bem com gotinhas de chocolate. Criamos alguns itens para nossas sobremesas saudáveis com coberturas opcionais:

```
var items = [ Item(name: "Eggplant Brownie", calories: 10),
               Item(name: "Zucchini Muffin", calories: 10),
               Item(name: "Cookie", calories: 10),
               Item(name: "Coconut oil", calories: 500),
               Item(name: "Chocolate frosting", calories: 1000),
               Item(name: "Chocolate chip", calories: 1000)
             ]
```

Implementamos os dois métodos de fonte de dados de uma tabela que vimos diversas vezes. O que retorna a quantidade de elementos:

```
func tableView(tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
    return items.count
}
```

E o que retorna o conteúdo de cada célula:

```
func tableView(tableView: UITableView,
               cellForRowAtIndexPath indexPath: NSIndexPath)
    -> UITableViewCell {

    let row = indexPath.row
    let item = items[ row ]
    var cell = UITableViewCell(style:
        UITableViewCellStyle.Default, reuseIdentifier: nil)
    cell.textLabel.text = item.name
    return cell
}
```

Se rodarmos a aplicação, notamos que a tabela já reflete os dados do array, mas ainda temos alguns passos pela frente: sempre é bom entendermos melhor o que fizemos, melhoramos o código e continuamos.

10.1 PROTOCOLOS DA API

Mas por que não estamos escrevendo mais `override func tableView` e, inclusive, se usarmos a palavra `override` o compilador reclama? Acontece que o protocolo `UITableViewDataSource` não fornece nenhuma implementação do método, somente sua cara, sua interface (basicamente seu nome e seus parâmetros).

Já ao herdar o `UITableViewController`, ele mesmo implementa o protocolo `UITableViewDataSource` e já escreve uma versão inicial desses métodos. Como no caso do `UITableViewController` estamos sobrescrevendo um método já existente, usamos a palavra-chave `override` para deixar isso claro. Como somente puxamos a definição do protocolo, não estamos sobrescrevendo nada. Você pode conferir o protocolo `UITableViewDataSource` mantendo o `command` apertado e clicando em seu nome:

```
protocol UITableViewDataSource : NSObjectProtocol {

    func tableView(tableView: UITableView,
        numberOfRowsInSection section: Int) -> Int

    func tableView(tableView: UITableView,
        cellForRowAtIndexPath indexPath: NSIndexPath)
        -> UITableViewCell

    // optional
}
```

A primeira vez que você executa esse clique o Xcode pode dizer que não encontrou o que procurara (`Symbol not found`). Em `XCode`, `Preferences`, `Downloads` você pode baixar a documentação do iOS.

Repare que o `UITableViewDataSource` possui dois métodos obrigatórios, justamente os dois que definimos.

Testamos nossa aplicação e ela já mostra a tabela com os alimentos que entram em nossa refeição. Falta agora ser capaz de selecioná-los.

10.2 SELEÇÃO MÚLTIPLA E CELL ACCESSORY

Cada vez que selecionamos um item, gostaríamos de marcá-lo com um `check`, algo que indique que vamos utilizá-lo na hora de criar nossa refeição. O `UITableViewCell` fornece uma maneira de colocar informações extras em uma célula através da enum `UITableViewCellAccessoryType*`. Por exemplo, o `UITableViewCellAccessoryType.Checkmark` adiciona uma marca de `check` enquanto o `UITableViewCellAccessoryType.None` é o padrão sem nenhuma marca.

ENUM

Uma enum é basicamente uma coleção de valores limitados e pré-fixados. Veja no caso a seguir, onde definimos o tipo de erro como possivelmente sendo `FATAL` ou `WARNING`:

```
enum ErrorType {  
    case FATAL  
    case WARNING  
}
```

Enums podem ser utilizadas em programação funcional como um recurso para garantir que todos os casos foram tratados, como quando fazemos *pattern matching*. Em Orientação a Objetos, é comum utilizarmos polimorfismo para obter resultados similares.

Precisamos escrever o método que é invocado toda vez que o usuário tenta selecionar uma célula:

```
func tableView(tableView: UITableView,  
    didSelectRowAtIndexPath indexPath: NSIndexPath) {  
}
```

Mas ao tentar escrever `tableView` percebemos que o editor não nos ajuda: esse método ainda não existe. Como assim?

Se vamos implementar os comportamentos de `delegate` como fizemos aqui, temos que dizer que estamos suportando esse protocolo, o `UITableViewDelegate`:

```
class ViewController: UIViewController,
    UITableViewDataSource, UITableViewDelegate {
    // ...
}
```

Agora sim o Xcode nos ajuda e podemos, dentro desse método, recuperar a célula que foi escolhida pelo usuário:

```
func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)!
}
```

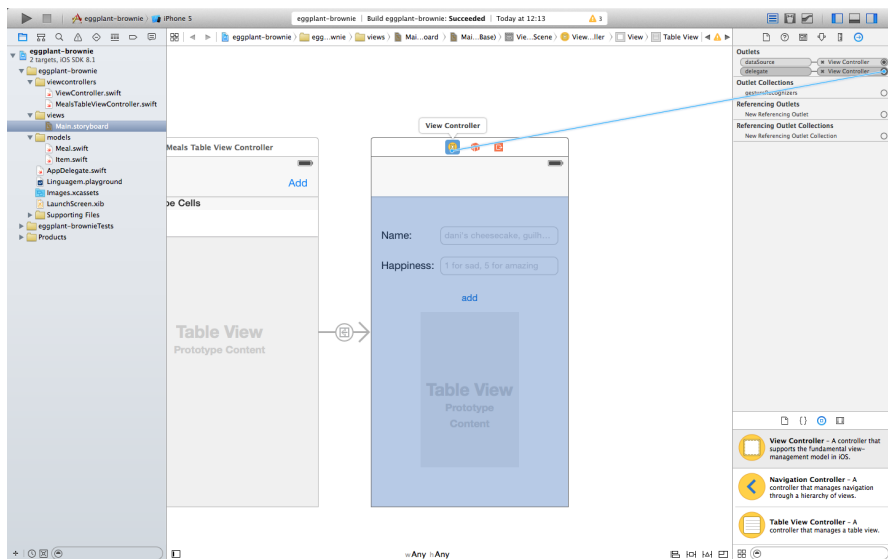
Marcamos colocando um marcador do tipo `Checkmark`:

```
func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)
    cell.accessoryType = UITableViewCellAccessoryType.Checkmark
}
```

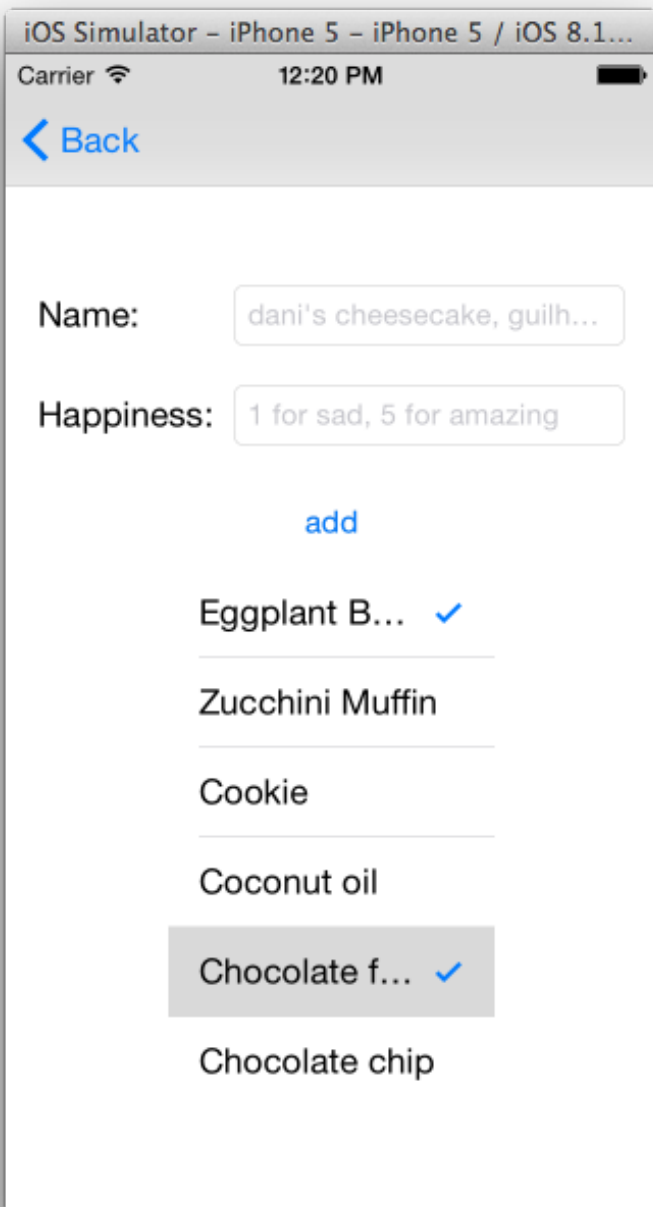
O código não compila, a tipagem explícita escondeu que a `cell` é opcional. Vamos verificar se a célula foi encontrada para então marcá-la:

```
func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)
    if cell == nil {
        return
    }
    cell!.accessoryType = UITableViewCellAccessoryType.Checkmark
}
```

Rodamos nossa aplicação, selecionamos os elementos nos quais temos interesse e nada acontece. Esquecemos de marcar que o nosso `ViewController` não só é responsável pela fonte de dados (`data source`) de nossa tabela, mas também terá seus métodos invocados quando algo na tabela acontecer. Ele é nosso *observer* da tabela - nosso *delegate*. Voltamos à edição visual da `TableView` e agora arrastamos o *delegate* para nosso `ViewController`, como fizemos para o `dataSource`:



Testamos nossa aplicação e temos o resultado:



10.3 DESSELECIONANDO ELEMENTOS

Opa, mas e se seleccionei errado? Já era? Faltou implementarmos suporte para que quem está marcado possa ser desmarcado. Verificamos se nossa célula não possui um `checkmark` e, se não possuir, o adicionamos; caso possua, removemos. Vamos adicionar essa verificação no mesmo método a que atribuímos o `Checkmark` anteriormente:

```
if (cell!.accessoryType == UITableViewCellAccessoryType.None) {
    cell!.accessoryType = UITableViewCellAccessoryType.Checkmark
} else {
    cell!.accessoryType = UITableViewCellAccessoryType.None
}
```

10.4 ARMAZENANDO A SELEÇÃO

Precisamos agora armazenar nossa seleção em algum lugar, para podermos criar nossa refeição na hora adequada. Podemos criar um array que mantém os números das linhas selecionadas:

```
var selected = Array<Int>()

func tableView(tableView: UITableView,
               didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)!

    if (cell.accessoryType == UITableViewCellAccessoryType.None){
        cell.accessoryType =
            UITableViewCellAccessoryType.Checkmark
        selected.append(indexPath.row)
    } else {
        cell.accessoryType = UITableViewCellAccessoryType.None
        // remove
    }
}
```

Mas um array de `Int`? Sério mesmo? Estamos trabalhando com objetos e agora brincamos com arrays de `Int`? Nosso array é de `Item`:

```
var selected = Array<Item>()

func tableView(tableView: UITableView,
               didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)!

    if (cell.accessoryType == UITableViewCellAccessoryType.None){
        cell.accessoryType =
            UITableViewCellAccessoryType.Checkmark
        selected.append(items[indexPath.row])
    } else {
        cell.accessoryType = UITableViewCellAccessoryType.None
        // remove
    }
}
```

Para testarmos, ao inicializarmos nosso `Meal` desejamos configurar seu `meal.items` e imprimi-lo:

```
@IBAction func add() {
    if nameField == nil || happinessField == nil {
        return
    }

    let name = nameField!.text
    let happiness = happinessField!.text.toInt()

    if happiness == nil {
        return
    }

    let meal = Meal(name: name, happiness: happiness)
    meal.items = selected
    println(
        "eaten: \(meal.name) \(meal.happiness) \(meal.items)")

    if delegate == nil {
        return
    }
}
```

```

        delegate!.add(meal)

        if let navigation = self.navigationController {
            navigation.popViewControllerAnimated(true)
        }
    }
}

```

Podemos testar nossa aplicação e ver o resultado: perfeito, ele imprime no log um array com o mesmo número de elementos que selecionei em minha tabela:



Falta removermos os desselecionados. Para isso, desejamos chamar a função `remove` de nosso array:

```
selected.remove....
```

Mas não existe função que recebe o objeto a ser removido. Somente uma função que requer a posição a ser removida:

```
selected.removeAtIndex(position)
```

Logo, devemos primeiro achar a posição de nosso elemento em nosso array, usando a função `find`:

```

let position = find(selected, items[indexPath.row])
selected.removeAtIndex(position)

```

Mas o código não compila. Acontece que a função `find` passa por todos os elementos de nosso array, verificando se cada um deles é igual (`==`) ao elemento que estamos procurando. Caso ele encontre o valor, ele retorna a posição, caso contrário, retorna `nil`. Uma implementação possível para essa função `find`, que já existe, seria algo similar a:

```
func find(elements:Array<Item>, toFind:Item) -> Int? {
    let max = elements.count - 1
    for i in 0...max {
        if toFind == elements[i] {
            return i
        }
    }
    return nil
}
```

Implementando o `==`

Note que, para encontrarmos o elemento, usamos o operador `=`, mas como comparar dois `Items`? Precisamos de alguma maneira falar que esses itens são comparáveis, que temos como verificar se eles são iguais, `Equatable`. Portanto, fazemos nossa classe adotar o protocolo `Equatable`:

```
class Item: Equatable {
    // code
}
```

E implementar a função `==`, que recebe dois itens e devolve um `Bool`:

```
func ==(first:Item, second:Item) -> Bool {
    return first.name == second.name &&
        first.calories == second.calories
}
```

Muito cuidado! A função `==` deve ser definida fora da classe:

```
// Item.swift

class Item: Equatable {
```

```
// code
}
func ==(first:Item, second:Item) -> Bool {
    return first.name == second.name &&
        first.calories == second.calories
}
```

A implementação do `==` permite que o programador faça comparações entre objetos ao utilizar diversas partes da API fornecida pelo iOS. Além disso, outros programadores criam suas bibliotecas baseadas na existência de uma comparação compatível com a definição do `==`.

Agora sim, podemos verificar se nosso item existe dentro de nosso array utilizando a função `find` que já existe. Mas lembre-se: o `find` retorna um `Optional`, vamos verificar se encontramos a posição com o elemento que queremos remover, e aí o removemos:

```
if let position = find(selected, items[indexPath.row]) {
    selected.removeAtIndex(position)
}
```

O código final do método vai permitir incluir ou remover um elemento selecionado em nosso array:

```
func tableView(tableView: UITableView,
               didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)
    if cell == nil {
        return
    }
    if (cell!.accessoryType ==
        UITableViewCellAccessoryType.None) {
        cell!.accessoryType =
            UITableViewCellAccessoryType.Checkmark
        selected.append(items[indexPath.row])
    } else {
        cell!.accessoryType = UITableViewCellAccessoryType.None
        if let position = find(selected, items[indexPath.row]) {
            selected.removeAtIndex(position)
        }
    }
}
```

```
    }  
}
```

Lembrando, nossa função de adicionar cria a refeição, seta os itens selecionados e os imprime:

```
@IBAction func add() {  
    if nameField == nil || happinessField == nil {  
        return  
    }  
  
    let name = nameField!.text  
    let happiness = happinessField!.text.toInt()  
  
    if happiness == nil {  
        return  
    }  
  
    let meal = Meal(name: name, happiness: happiness)  
    meal.items = selected  
    println(  
        "eaten: \(meal.name) \(meal.happiness) \(meal.items)"  
    )  
  
    if delegate == nil {  
        return  
    }  
    delegate!.add(meal)  
  
    if let navigation = self.navigationController {  
        navigation.popViewControllerAnimated(true)  
    }  
}
```

Testamos nossa aplicação e agora somos capazes de comer `cookies` de óleo de coco, selecionar e desselecionar diversos itens e, ao adicionar a refeição final, temos no log o resultado com a impressão de quantos itens estavam em nossa refeição:

```
eaten: cookie de oleo de coco 5  
[eggplant_brownie.Item, eggplant_brownie.Item, eggplant_brownie.Item]
```

All Output ▾   

10.5 RESUMO

Vimos como podemos adicionar uma `TableView` em um `controller` existente e conectá-la ao seu `data source` e seu `delegate` usando o *pattern* `Observer`. Vimos também como receber eventos de seleção em nosso `delegate` e com isso utilizar a enum `UITableViewCellAccessoryType` para marcar as células como selecionadas. Armazenamos as células selecionadas e removemos as desselecionadas de um array que reflete tais escolhas do usuário.

Para a remoção, foi importante conhecer mais de um protocolo bastante utilizado para usar com coleções: a capacidade de dois itens serem comparados com o `==`, o protocolo `Equatable`. Utilizamos a função `find` e a `removeAtPosition` para buscar e remover um elemento de nosso array.

Por fim, usamos esse array logo após instanciar nossa refeição para preencher os campos que foram escolhidos.

CAPÍTULO 11

Criando novos itens

Chegou a hora de permitir ao usuário criar novos itens. Para isso, adicionaremos um novo `ViewController` que representará um formulário. Ele terá o campo com o nome e o número de calorias desse item, além de um botão de adicionar. Ao concluir a operação no formulário, devemos voltar para a tela de edição da refeição.

Já conhecemos tudo isso, portanto veremos uma nova maneira de realizar o `push`, além de aproveitar a oportunidade para exercitar tudo o que fizemos anteriormente. Citamos anteriormente um dos problemas de manutenção de código ao lidar com diversos `segues` visuais, o código pode ficar cheio de cláusulas do tipo `if`, com uma complexidade alta.

Uma outra alternativa, muito citada por desenvolvedores sêniores, está ligada à navegação entre telas de maneira programática. Da mesma maneira como fizemos um `pop` para desempilhar uma tela, utilizaremos um `push` (ou similar) para empilhar uma tela.

Nosso primeiro passo é criar a representação visual de nosso `viewController`, mas espere. Dessa vez não puxaremos o `controller` para nosso `storyboard`.

BOA PRÁTICA OU CODE SMELL? O STORYBOARD É UMA SOLUÇÃO DO BEM OU DO MAL?

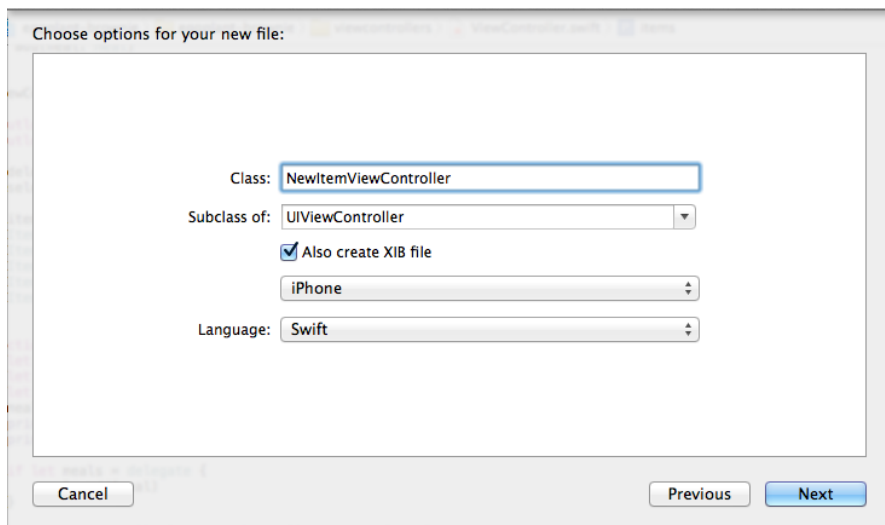
O *Storyboard* veio como alternativa para a criação de cada view separadamente, ele também facilita o arrasta e solta como maneira de programar a iteração entre views. Por um lado, esse trabalho fica muito simples; por outro, ele acaba forçando a criação de código com determinadas práticas que não são consideradas as melhores (uma série de `ifs`, por exemplo).

Além disso, com o passar do tempo o storyboard pode ficar com tantas telas que fica difícil manter todas as conexões visualmente compreensíveis.

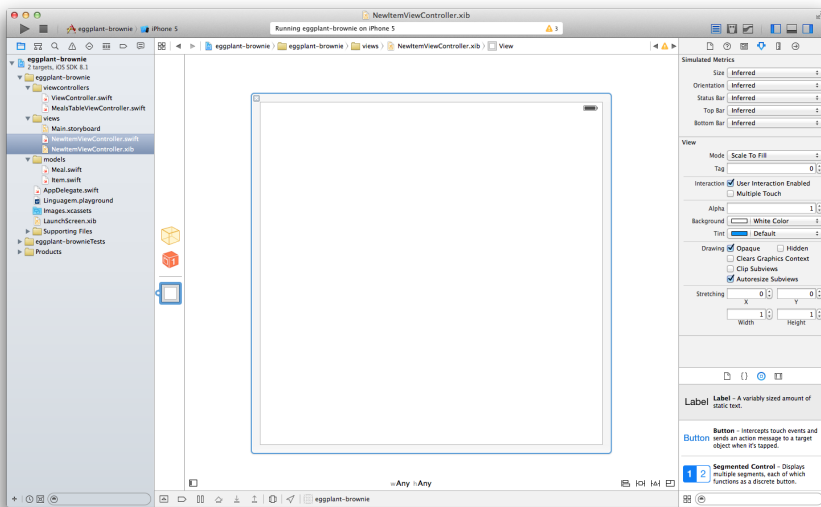
O maior problema do storyboard acontece quando trabalhamos em equipe: quando duas pessoas o alteram de maneira incompatível. Será o trabalho de um deles de fazer o processo de *merge* na mão. Essa pessoa deverá alterar um `xml` que descreve as cenas, pois o editor visual não será capaz de mostrar as diferenças.

Como regra geral, em projetos mais simples, com menos views, o uso do storyboard pode não comprometer a manutenção de seu projeto. Aqui aprenderemos tanto a maneira visual com o storyboard quanto a programática com arquivos `Xib` (antigos `nib`).

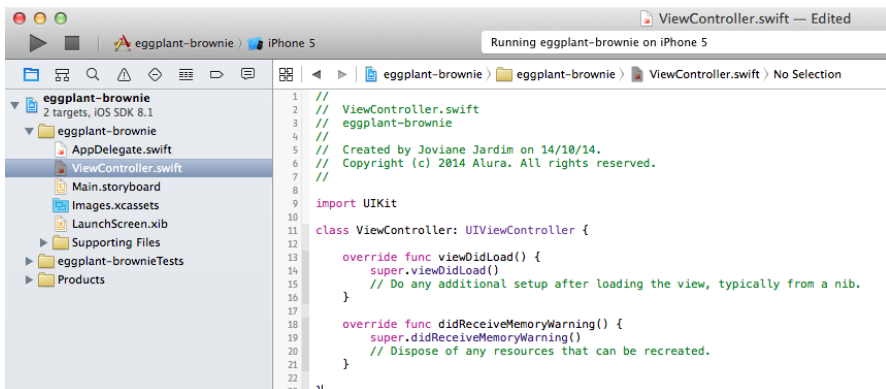
Vamos então isolar e criar um arquivo visual separado para esse nosso `viewController`, o arquivo visual (nas versões mais recentes do iOS) possui a extensão `XIB`. Portanto, no grupo `views`, escolhemos o menu `File, New, iOS, Source, Cocoa Touch Class, NewItemViewController, UIViewController, Also create XIB File e Swift`:



Ele é criado no nosso grupo de `views`. Note que temos agora dois arquivos, o XIB:



E o controller que já conhecemos:



Como tanto o controller quanto o xib foram criados no mesmo diretório, vamos mover o viewController da mesma maneira como fizemos

anteriormente, movendo diretamente no diretório, removendo a referência e adicionando novamente pelo Xcode.

Em nosso `XIB`, selecione sua `View` clicando no ícone branco à esquerda e escolha o tamanho do iPhone que estamos utilizando. Adicionamos agora os campos do formulário de um `item`, que são `name` e `calories`, e um botão de confirmar inclusão, chamado `add`:

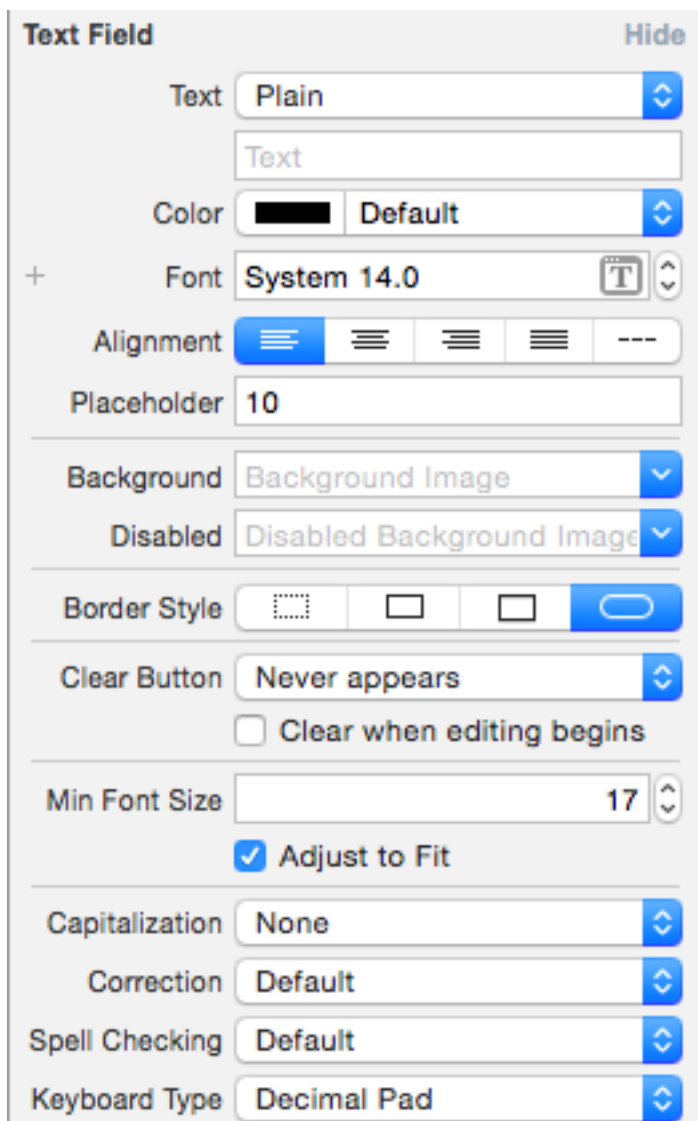
158

Name:

Calories:

add

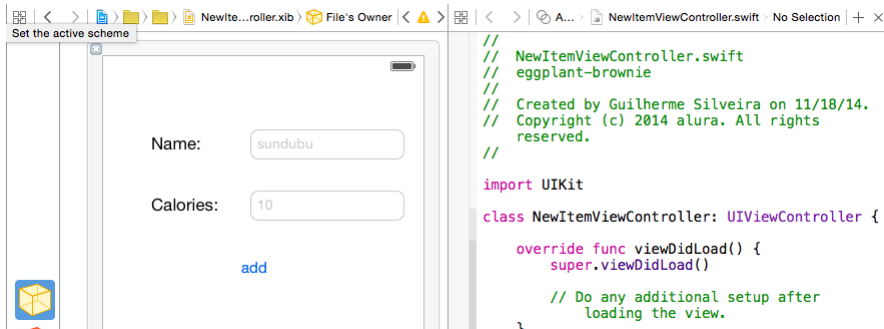
Lembre-se de mudar o teclado do campo `calories` para `Decimal` e colocar *placeholder* nos dois campos: `sundubu` e `10`.



The image shows the SwiftUI **Text Field** inspector in Xcode. The title bar includes "Text Field" and a "Hide" button. The settings are as follows:

- Text:** Plain (dropdown)
- Color:** Default (dropdown)
- Font:** System 14.0 (dropdown)
- Alignment:** Left (selected)
- Placeholder:** 10 (text field)
- Background:** Background Image (dropdown)
- Disabled:** Disabled Background Image (dropdown)
- Border Style:** Rounded (selected)
- Clear Button:** Never appears (dropdown)
- ☐ Clear when editing begins
- Min Font Size:** 17 (dropdown)
- ☒ Adjust to Fit
- Capitalization:** None (dropdown)
- Correction:** Default (dropdown)
- Spell Checking:** Default (dropdown)
- Keyboard Type:** Decimal Pad (dropdown)

No XIB, conferimos que ao selecionar o `file owner` sua identidade é o nosso `NewItemViewController`. Podemos até usar o `assistant editor` para conferir que a conexão ainda está feita:



Na nossa tela de adicionar nova refeição, dentro do `storyboard`, vamos agora colocar o botão para conseguirmos fazer a navegação para esta tela nova. Poderíamos fazer a criação deste botão de forma visual, arrastando o componente para a tela direto no `storyboard`, porém desta vez faremos programaticamente.

Abrimos nosso `ViewController`. Dentro dele, sobrescrevemos o método `viewDidLoad`, onde criaremos um `UIBarButtonItem`. Para criarmos um `UIBarButtonItem`, precisamos dizer qual o texto que queremos no botão (`title`), qual o formato (`style`), quem deve ser chamado ao clicarmos nele (`target`) e qual a ação que queremos executar neste objeto que for chamado (`action`).

```
override func viewDidLoad() {
    let newItemButton = UIBarButtonItem(title: "new item",
        style: UIBarButtonItemStyle.Plain,
        target: self,
        action: ?????)
}
```

Mas como podemos passar a chamada de um método para o botão executar? Não podemos chamar o método direto, pois desse modo ele será exe-

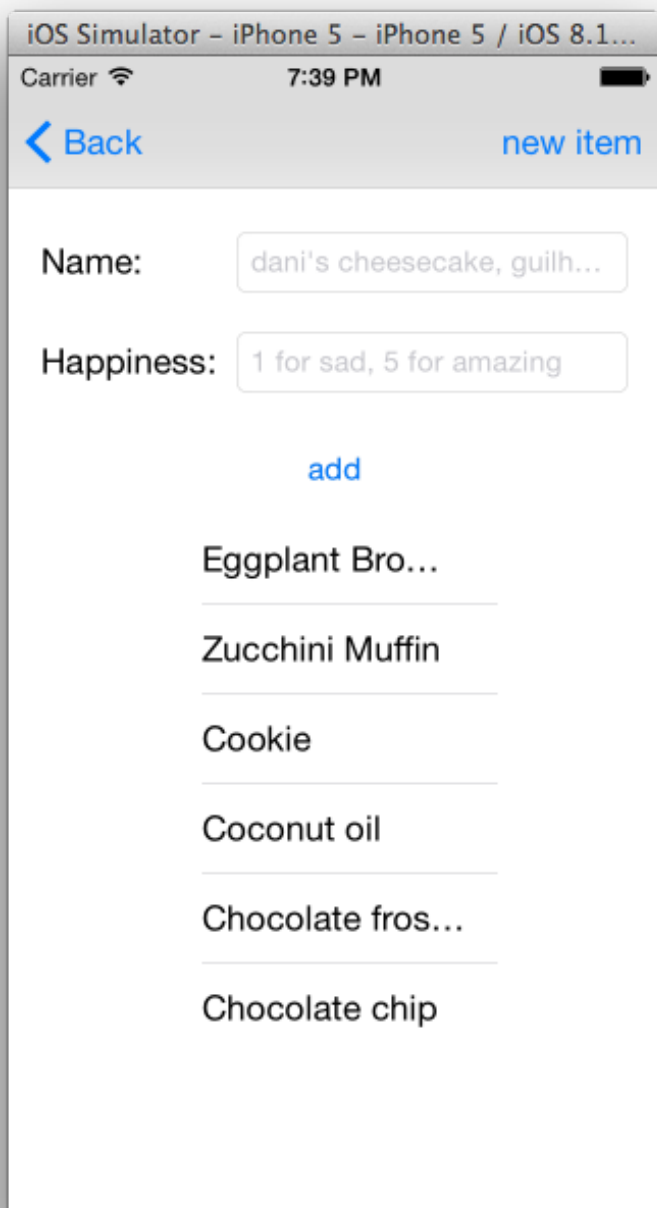
cutado no momento em que o chamarmos. Precisamos que o botão chame o método. Para isto, criamos um `Selector`, passando como parâmetro o nome do método que queremos que seja chamado:

```
override func viewDidLoad() {  
    let newItemButton = UIBarButtonItem(title: "new item",  
                                        style: UIBarButtonItemStyle.Plain,  
                                        target: self,  
                                        action: Selector("showNewItem"))  
}
```

Adicionamos também o botão no lado direito de nosso `Navigation Controller`:

```
override func viewDidLoad() {  
    let newItemButton = UIBarButtonItem(title: "new item",  
                                        style: UIBarButtonItemStyle.Plain,  
                                        target: self,  
                                        action: Selector("showNewItem"))  
    navigationItem.rightBarButtonItem = newItemButton  
}
```

Rodamos a aplicação e podemos ver que o botão foi criado com sucesso:



Criamos o código da função que abrirá a tela de `new item` em nosso `ViewController`, imprimindo uma mensagem de teste:

```
@IBAction func showNewItem() {
    println("new item")
}
```

E testamos nossa aplicação, o botão aparece, e ao clicarmos nele vemos a mensagem de log. Perceba o poder de criar views programaticamente: você pode customizar todas as características de seu programa dinamicamente. Vale como regra geral desenhar o que é fixo via arrasta e solta (*interface builder*); e o que é customizado, dinâmico, acabamos fazendo programaticamente, como por exemplo um botão que só aparece quando o usuário está logado.

11.1 PUSHANDO VIEWS PARA A PILHA PROGRAMATICAMENTE

Agora estamos prontos para pegar nosso navegador e mostrar a tela `NewItemViewController`:

```
@IBAction func showNewItem() {
    let newItem = ???
    if let navigation = navigationController {
        navigation.pushViewController(newItem, animated: true)
    }
}
```

Precisamos criar nosso `controller`, mas lembre-se que estamos programando orientado a objetos e nosso `NewItemViewController` é uma classe: basta instanciá-la!

```
@IBAction func showNewItem() {
    let newItem = NewItemViewController()
    if let navigation = navigationController {
        navigation.pushViewController(newItem, animated: true)
    }
}
```

Testamos e, ao clicar no botão, temos uma tela preta!

```

override func viewDidLoad() {
    let newItemButton = UIBarButtonItem(title: "Add",
                                        style: UIBarButtonItemStyle.Plain,
                                        target: self,
                                        action: Selector("showNewItem"))
    navigationItem.rightBarButtonItem = newItemButton
}

@IBAction func showNewItem() {
    let newItem = NewItemViewController()
    if let navigation = navigationController {
        navigation.pushViewController(newItem, animated: true)
    }
}

@IBAction func add() {
    if nameField == nil || happinessField == nil {
        return
    }

    let name = nameField!.text
    let happiness = happinessField!.text.toInt()
    if happiness == nil {
        return
    }

    let meal = Meal(name: name, happiness: happiness)
    meal.items = selected
    println("eaten: \(meal.name) \(meal.happiness)")

    if delegate == nil {
        return
    }

    delegate!.add(meal)
}

```



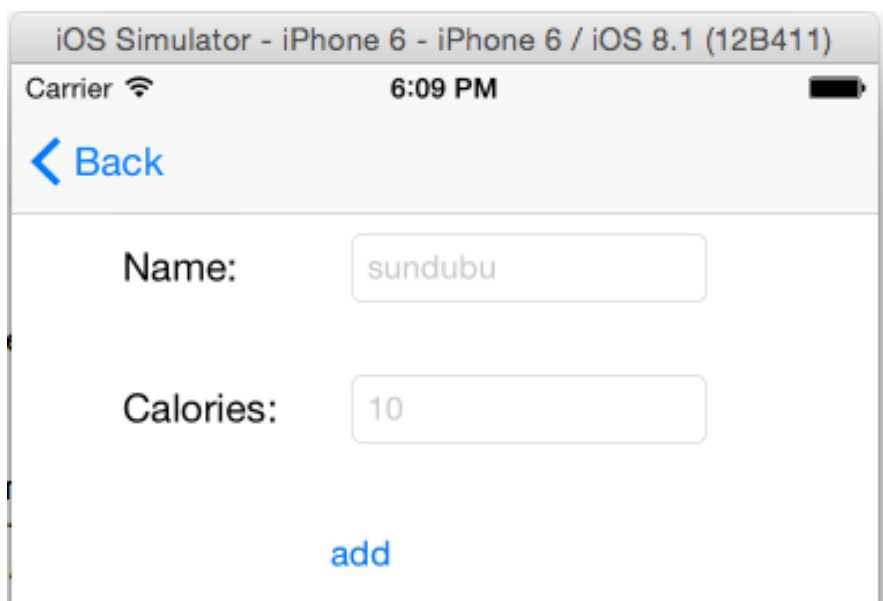
O que acontece? Nós pedimos para instanciar o `NewItemViewController` mas ninguém falou qual arquivo `xib` era para ser usado. O `xib` é a nova versão do arquivo `nib`, portanto, ao construirmos nosso `NewItemViewController`, dizemos qual arquivo de view queremos que ele leia:

```

@IBAction func showNewItem() {
    let newItem = NewItemViewController(
        nibName: "NewItemViewController", bundle: nil)
    if let navigation = navigationController {
        navigation.pushViewController(newItem, animated: true)
    }
}

```

Testamos a aplicação e, mesmo sem criar o `segue`, fomos capazes de manipular a tela atual! Note que, se fosse importante passar algum argumento para nosso `controller`, bastaria chamar o construtor passando o argumento a mais (e criá-lo em nossa classe, claro), algo muito mais educado do que ficar setando propriedades após a construção.



11.2 VOLTANDO DE UM PUSH PROGRAMÁTICO

Precisamos agora voltar para a nossa tela anterior ao clicar no botão de confirmação. Já conhecemos o código do `pop`, logo, em nosso `NewItemViewController.swift` colocamos a função `addNewItem`. Além disso, definiremos as duas variáveis que nos ajudarão a ler os campos do item novo:

```
class NewItemViewController: UIViewController {  
    @IBOutlet var nameField:UITextField?  
    @IBOutlet var caloriesField:UITextField?
```

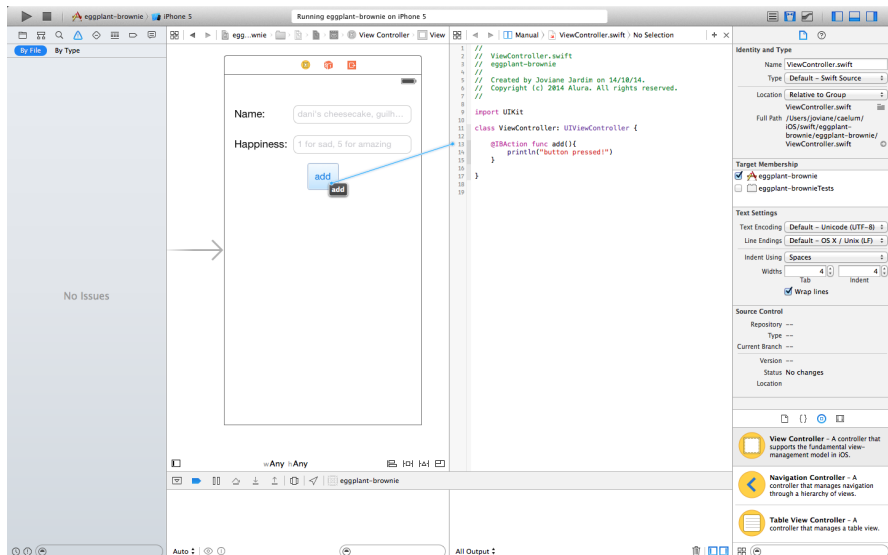
```

@IBAction func addNewItem() {
}

}

```

Usamos o arrasta da bolinha para conectarmos os *outlets* e a *action* com os campos de texto e o botão.



Executamos o `pop`:

```

@IBOutlet var nameField:UITextField?
@IBOutlet var caloriesField:UITextField?

@IBAction func addNewItem() {
    if let navigation = navigationController {
        navigation.popViewControllerAnimated(true)
    }
}

```

Agora lemos os valores dos campos e criamos um novo item:

```
@IBAction func addNewItem() {
    let name = nameField.text
    let calories = caloriesField.text

    let item = Item(name: name, calories: calories)
    if let navigation = navigationController {
        navigation.popViewControllerAnimated(true)
    }
}
```

Algumas coisas não compilam, primeiro devemos verificar valores opcionais:

```
@IBAction func addNewItem() {
    if nameField == nil || caloriesField == nil {
        return
    }
    let name = nameField!.text
    let calories = caloriesField!.text

    let item = Item(name: name, calories: calories)
    if let navigation = navigationController {
        navigation.popViewControllerAnimated(true)
    }
}
```

E agora, `calories` ainda é `String`! Queremos converter para `Double`, procuramos o `toDouble` e vemos que ele não existe na classe de `String` padrão. Isso pois conversão de ponto decimal está ligada à localização e internacionalização. No nosso caso, usaremos o padrão da linguagem Objective C, que já possuía uma `String` (`NSString`) com o método de conversão. Criamos a instância de uma `NSString` e convertemos:

```
@IBAction func addNewItem() {
    if nameField == nil || caloriesField == nil {
        return
    }
}
```

```

    let name = nameField!.text
    let calories =
        NSString(string: caloriesField!.text).doubleValue

    let item = Item(name: name, calories: calories)
    if let navigation = navigationController {
        navigation.popViewControllerAnimated(true)
    }
}

```

Testamos a aplicação e ela vai e vem, mas ainda não notificamos a tela para a qual voltamos com o dado do novo item que acaba de ser adicionado. Está na hora de fazer nosso `delegate`.

11.3 INVOCANDO UM DELEGATE DE FORMA PROGRAMÁTICA

Assim como fizemos antes, queremos notificar a tela que nos invocou de que o trabalho foi finalizado e temos um novo item para adicionar nela. Vamos criar o protocolo de nosso `delegate`, o `AddAnItemDelegate`:

```

protocol AddAnItemDelegate {
    func addNew(item: Item)
}

```

E o implementamos, adotamos o `AddAnItemDelegate` em nosso `ViewController`:

```

class ViewController: UIViewController, UITableViewDataSource,
    UITableViewDelegate, AddAnItemDelegate {

    // ...

    func addNew(item: Item) {
        items.append(item)
    }

    // ...
}

```

```
}
```

Precisamos agora atualizar nossa tabela, mas onde está nossa `tableView`? Quando o `controller` era do tipo `UITableViewController` era fácil: ele já vinha com uma tabela. Agora precisamos criar um novo outlet e conectá-lo a nossa tabela:

```
@IBOutlet var tableView: UITableView?
```

```
func addNew(item: Item) {  
    items.append(item)  
    if tableView == nil {  
        return  
    }  
    tableView!.reloadData()  
}
```

Não esquecemos de conectar o outlet usando o *drag and drop*.

Como trabalhar agora no nosso `NewItemViewController`? Precisamos de um `delegate` lá, logo, como um bom cidadão, recebemo-lo em nosso inicializador:

```
class NewItemViewController: UIViewController {  
  
    let delegate: AddAnItemDelegate?  
    init(delegate: AddAnItemDelegate?) {  
        self.delegate = delegate  
    }  
    // ...  
}
```

Mas o compilador reclama. Ao definirmos nosso inicializador, precisamos invocar o inicializador de nossa classe pai. Com certeza! Antes, invocávamos o construtor que recebia o `nibName`. O que era o `nibName`? O nome do arquivo que representa a `view`, a cena conectada a este `viewController`, então vamos invocá-lo logo após configurar nosso `delegate`:

```
class NewItemViewController: UIViewController {

    let delegate:AddAnItemDelegate?
    init(delegate:AddAnItemDelegate) {
        self.delegate = delegate
        super.init(nibName: "NewItemViewController", bundle: nil)
    }
    // ...
}
```

E novamente o compilador reclama: dessa vez ele pede a criação de um inicializador que receba um `NSCoder`. Acontece que a classe pai, `UIViewController` **requer** tal inicializador, portanto, vamos redefini-lo, simplesmente invocando o construtor de nosso pai:

```
class NewItemViewController: UIViewController {

    let delegate:AddAnItemDelegate?
    init(delegate:AddAnItemDelegate) {
        self.delegate = delegate
        super.init(nibName: "NewItemViewController", bundle: nil)
    }

    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }
    // ...
}
```

NOMENCLATURA DE PARÂMETRO

Mas que história é essa? Um parâmetro com dois nomes? O nome interno do nosso parâmetro é `aDecoder`, enquanto quem invoca nosso inicializador usará o nome `coder`.

O exemplo a seguir demonstra a utilização de um nome interno e externo para deixar claro o valor que estamos referenciando:

```
class User {
    init(name newName:String) {
        println("creating a \(newName)")
    }
}

let guilherme = User(name: "guilherme")
```

Claro, agora precisamos invocar o `delegate` para adicionar um elemento:

```
@IBAction func addNewItem() {
    if nameField == nil || caloriesField == nil {
        return
    }
    let name = nameField!.text
    let calories =
        NSString(string: caloriesField!.text).doubleValue

    let item = Item(name: name, calories: calories)

    if delegate == nil {
        return
    }
    delegate!.addNew(item)

    if let navigation = navigationController {
        navigation.popViewControllerAnimated(true)
    }
}
```

```
    }
}
```

Pronto. Na prática, o que acontece é que invocaremos o construtor em que estamos interessados, alterando nossa antiga chamada de inicialização para invocar o que recebe um `delegate`, portanto em nosso `ViewController`:

```
@IBAction func showNewItem() {
    let newItem = NewItemViewController(delegate: self)
    if let navigation = navigationController {
        navigation.pushViewController(newItem, animated: true)
    }
}
```

Nosso código do `ViewController` fica assim:

```
import UIKit

protocol AddAMealDelegate {
    func add(meal: Meal)
}

class ViewController: UIViewController, UITableViewDataSource,
    UITableViewDelegate, AddAnItemDelegate {

    var items = [ Item(name: "Eggplant Brownie", calories: 10),
        Item(name: "Zucchini Muffin", calories: 10),
        Item(name: "Cookie", calories: 10),
        Item(name: "Coconut oil", calories: 500),
        Item(name: "Chocolate frosting", calories: 1000),
        Item(name: "Chocolate chip", calories: 1000)
    ]

    @IBOutlet var nameField: UITextField!
    @IBOutlet var happinessField: UITextField!
    var delegate:AddAMealDelegate?
    var selected = Array<Item>()

    @IBOutlet var tableView: UITableView?

    func addNew(item: Item) {
```

```
        items.append(item)
        if tableView == nil {
            return
        }
        tableView!.reloadData()
    }

func tableView(tableView: UITableView,
               numberOfRowsInSection section: Int) -> Int {
    return items.count
}

func tableView(tableView: UITableView,
               cellForRowAtIndexPath indexPath: NSIndexPath)
               -> UITableViewCell {

    let row = indexPath.row
    let item = items[ row ]
    var cell = UITableViewCell(style:
        UITableViewCellStyle.Default, reuseIdentifier: nil)
    cell.textLabel.text = item.name
    return cell
}

func tableView(tableView: UITableView,
               didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let cell = tableView.cellForRowAtIndexPath(indexPath)
    if cell == nil {
        return
    }
    if (cell!.accessoryType ==
        UITableViewCellAccessoryType.None) {
        cell!.accessoryType =
            UITableViewCellAccessoryType.Checkmark
        selected.append(items[indexPath.row])
    } else {
        cell!.accessoryType =
            UITableViewCellAccessoryType.None
        if let position =
```

```
        find(selected, items[indexPath.row]) {
            selected.removeAtIndex(position)
        }
    }
}

override func viewDidLoad() {
    let newItemButton = UIBarButtonItem(title: "new item",
                                        style: UIBarButtonItemStyle.Plain,
                                        target: self,
                                        action: Selector("showNewItem"))
    navigationItem.rightBarButtonItem = newItemButton
}

@IBAction func showNewItem() {
    let newItem = NewItemViewController(delegate: self)
    if let navigation = navigationController {
        navigation.pushViewController(newItem, animated:true)
    }
}

@IBAction func add() {
    if nameField == nil || happinessField == nil {
        return
    }

    let name = nameField!.text
    let happiness = happinessField!.text.toInt()
    if happiness == nil {
        return
    }

    let meal = Meal(name: name, happiness: happiness!)
    meal.items = selected
    println(
        "eaten: \(meal.name) \(meal.happiness) \(meal.items)")

    if delegate == nil {
```

```
        return
    }

    delegate!.add(meal)

    if let navigation = self.navigationController {
        navigation.popViewControllerAnimated(true)
    }
}

}
```

Enquanto isso, nosso `NewItemViewController`:

```
import UIKit

protocol AddAnItemDelegate {
    func addNew(item: Item)
}

class NewItemViewController: UIViewController {
    let delegate: AddAnItemDelegate?
    init(delegate: AddAnItemDelegate?) {
        self.delegate = delegate
        super.init(nibName: "NewItemViewController", bundle: nil)
    }
    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }

    @IBOutlet var nameField: UITextField?
    @IBOutlet var caloriesField: UITextField?

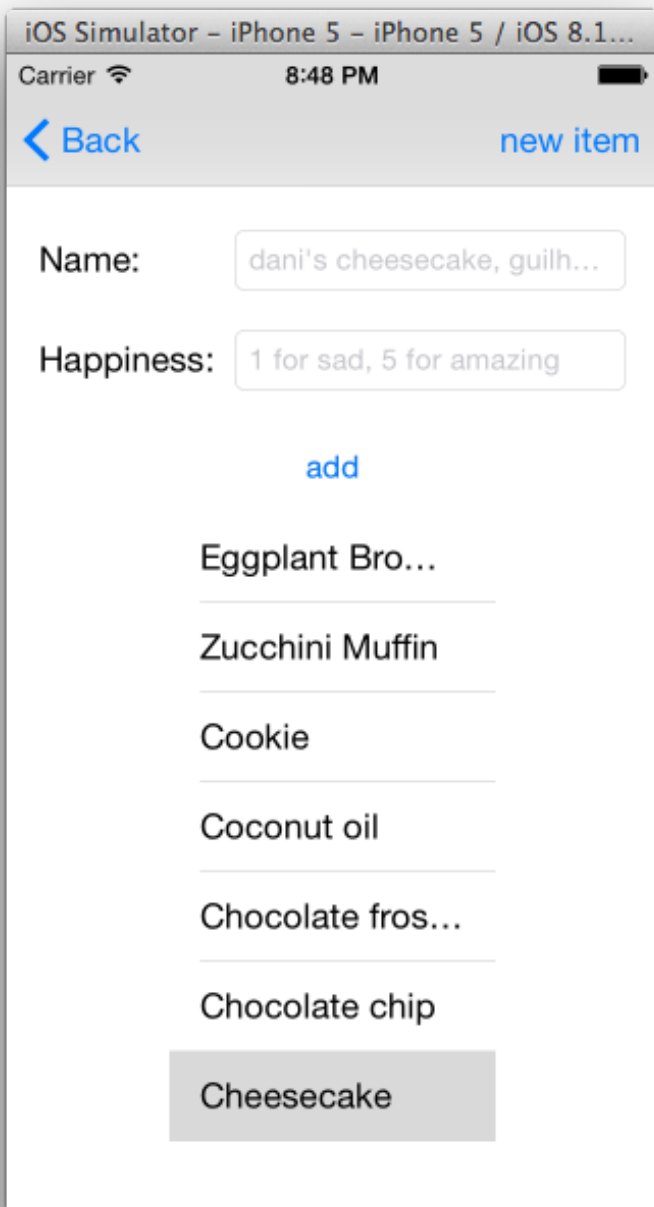
    @IBAction func addNewItem() {
        if nameField == nil || caloriesField == nil {
            return
        }
        let name = nameField!.text
        let calories =
            NSString(string: caloriesField!.text).doubleValue
    }
}
```

```
        let item = Item(name: name, calories: calories)

        if delegate == nil {
            return
        }
        delegate!.addNew(item)

        if let navigation = navigationController {
            navigation.popViewControllerAnimated(true)
        }
    }
}
```

Testamos nossa aplicação e, à medida que adicionamos novos itens, já os temos na nossa lista de itens a serem utilizados para a criação de uma refeição.



11.4 RESUMO

Vimos neste capítulo como criar um `XIB`, uma `view` de um `ViewController` que pode ser reutilizada mais facilmente, e vimos como definir o que nosso construtor recebe durante sua construção. Revisamos a criação de um formulário, o `push` e o `pop` programático, além do uso de um `delegate` para notificar nosso *observer* de uma tarefa executada em nossa tela.

CAPÍTULO 12

Mostrando os detalhes de uma refeição com Long press

Mas após adicionada, como conferimos os detalhes de cada refeição? Quais os itens e suas respectivas calorias?

Para fazermos isso, usaremos um recurso que frequentemente aparece em aplicativos iOS, o `Long Press`: ao segurarmos o clique em uma linha de nossa tabela, queremos visualizar a tela de detalhes.

Precisamos primeiramente dizer que nossa célula consegue reconhecer este tipo de interação e qual ação deve ocorrer quando for identificado o evento de `Long Press`. O responsável por fazer esse reconhecimento é o “elemento visual reconhecedor de movimento de manter apertado”, literalmente o `UILongPressGestureRecognizer`. Lembra quando criamos nossa célula no `MealsTableViewController`?

```

override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) ->
    UITableViewCell {
    let row = indexPath.row
    let meal = meals[ row ]

    var cell = UITableViewCell(
        style: UITableViewCellStyle.Default,
        reuseIdentifier: nil)
    cell.textLabel.text = meal.name
    return cell
}

```

Instanciamos o `UILongPressGestureRecognizer` e o adicionamos à nossa célula:

```

override func tableView(tableView: UITableView,
    cellForRowAtIndexPath
    indexPath: NSIndexPath) -> UITableViewCell {
    //...
    let longPress =
        UILongPressGestureRecognizer(/* parameters */)
    cell.addGestureRecognizer(longPress)
    return cell
}

```

Mas qual será o método executado quando o usuário mantiver o dedo pressionado nesta célula? De alguma maneira, devemos falar nosso alvo, o objeto a ser notificado (`target`) e o método a ser invocado (`action`). Já conhecemos esse par, o objeto e o `Selector`:

```

override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) ->
    UITableViewCell {
    //...
    let longPress = UILongPressGestureRecognizer(target: self,
        action: Selector("showDetails"))
    cell.addGestureRecognizer(longPress)
    return cell
}

```

Criamos o método para mostrar mais detalhes da refeição:

```
func showDetails(){  
}
```

Mas como saberemos qual refeição foi selecionada? Felizmente, o próprio `UILongPressGestureRecognizer` guarda a informação sobre em qual `view` o evento ocorreu, temos apenas que pedir o reconhecedor por parâmetro:

```
func showDetails(recognizer: UILongPressGestureRecognizer){  
  
}
```

Só precisamos agora indicar ao `LongPressGestureRecognizer` que queremos recebê-lo no momento em que o método `showDetails` for invocado. Para representarmos que queremos receber um parâmetro no nosso `Selector`, colocamos ":" (dois pontos) no final do nome do método. Isto é herança do Objective-C, onde indicávamos a passagem de parâmetros com os ":".

```
override func tableView(tableView: UITableView,  
                        cellForRowAtIndexPath indexPath: NSIndexPath) ->  
                        UITableViewCell {  
    //...  
    let longPress = UILongPressGestureRecognizer(target: self,  
                                                action: Selector("showDetails:"))  
    cell.addGestureRecognizer(longPress)  
    return cell  
}
```

Vamos mostrar os detalhes de nossa refeição, mas em qual momento queremos que eles apareçam? Quando o evento de `long press` começou! Portanto, verificamos o estado e colocamos um `println` para sabermos que funcionou.

```
func showDetails(recognizer: UILongPressGestureRecognizer){  
    if recognizer.state == UIGestureRecognizerState.Began {  
        println("Long press")  
    }  
}
```

```
}
}
```

Vamos rodar. Ao efetuarmos o `Long Press`, clicando e segurando a linha da tabela da qual queremos ver mais detalhes, o log mostra a mensagem:



12.1 RECUPERANDO ONDE OCORREU UM EVENTO DE LONG PRESS

Agora temos que recuperar qual a célula em que ocorreu o `Long Press` e o seu conteúdo. Para isso, primeiro extraímos do `recognizer` qual a `view` à qual ele estava atrelado, nossa célula:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view

        println("Long press")
    }
}
```

Agora que temos a célula, podemos buscar o índice dela em nossa `tableView` através do método `indexPathForCell`:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view
```

```
        let indexPath = tableView.indexPathForCell(cell)

        println("Long press")
    }
}
```

Mas, ao digitar a chamada para o método, o Xcode não consegue reconhecer o método pois a `view` não é nosso `UITableViewCell`, afinal, um `UILongPressGestureRecognizer` pode ser aplicado para qualquer tipo de `UIView`! Como nós, desenvolvedores, temos certeza de que temos uma célula aí dentro referenciada pela `view`, podemos fazer o `cast` para `UITableViewCell`:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)

        println("Long press")
    }
}
```

CODE SMELL: CAST

Sempre que fazemos um *cast* estamos dizendo que nós, como desenvolvedores, sabemos o que está acontecendo, e o compilador não. Se isso acontece, estamos correndo risco. O compilador existe para garantir diversas coisas, dentre elas, que não acessamos algo pensando que é outra coisa. O *casting* permite cometer esse erro (e estourar a aplicação).

Como fugir do cast? Em casos similares como esse, caso o `UILongPressGestureRecognizer` fosse genérico mas permitisse dizer o tipo de classe para o qual fosse aplicado, definiríamos em sua criação que ele só pode trabalhar com `UITableViewCell` e teríamos certeza de que as views onde ele ocorreu são do tipo `UITableViewCell`. Esse tipo de comportamento pode ser alcançado com o uso de *generics*, mas como o mesmo é um componente da biblioteca padrão do iOS, não temos como alterar o código para tal benefício de compilação.

Com a `cell` e uma `tableView` em mãos, somos capazes de descobrir em qual linha ocorreu a ação, mas repare que a IDE adiciona um `?` de opcional:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        let row = indexPath?.row

        println("Long press")
    }
}
```

Não queremos opcional e correr risco, logo, `if` nele:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
```

```
        if indexPath == nil {
            return
        }
        let row = indexPath!.row

        println("Long press")
    }
}
```

Portanto, podemos extrair a refeição que foi clicada, buscando-a em nosso array:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        println("Long press")
    }
}
```

Vamos mudar o `println` para mostrar o nome e o nível de felicidade da refeição selecionada:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        println("meal: \(meal.name) \(meal.happiness)")
    }
}
```

```
    }
}
```

Rodamos e agora o log mostra a refeição selecionada ao efetuarmos o Long Press:



Estamos prontos para nosso próximo passo: mostrar os dados na tela através de um outro `Controller`.

12.2 MOSTRANDO OS DETALHES EM UM ALERTA

Para mostrarmos os detalhes de uma refeição selecionada, criaremos um alerta, um `UIAlertController`, ou seja, um *popup* com os dados que queremos mostrar:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        let details = UIAlertController(/* parameters */)
    }
}
```

Quais parâmetros devemos passar ao nosso alerta? Primeiro, o título com o nome da refeição e a mensagem com o nível de felicidade:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        let details = UIAlertController(title: meal.name,
                                       message: "Happiness: \(meal.happiness)",
                                       /* extra parameter */ )
    }
}
```

Mas devemos falar também qual o tipo de alerta que desejamos mostrar, o estilo tradicional:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        let details = UIAlertController(title: meal.name,
                                       message: "Happiness: \(meal.happiness)",
                                       preferredStyle: UIAlertControllerStyle.Alert)
    }
}
```

Por fim, pedimos para mostrar nosso `controller` de alerta através do método `presentViewController`, animado. Não utilizamos o último pa-

râmetro para esse método, então o desejamos vazio:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        let details = UIAlertController(title: meal.name,
                                         message: "Happiness: \(meal.happiness)",
                                         preferredStyle: UIAlertControllerStyle.Alert)
        presentViewController(
            details, animated: true, completion: nil)
    }
}
```



Mas e como fechamos os detalhes? Precisamos avisar nossa tela de que queremos voltar para a tela anterior e remover a tela atual.

MODAL

Repare que, anteriormente, quando precisamos apresentar uma nova tela, utilizamos a navegação tradicional, com `push` para mostrar a tela e `pop` para fechar. Fizemos isso pois queríamos manter sempre o histórico de navegação, por quais telas passamos e para onde voltaríamos. No caso do `ActionController`, queremos apenas mostrar o popup, sem precisar interferir no fluxo de navegação anterior, portanto este popup não precisa entrar na pilha de navegação. Para estes casos, fazemos a navegação de modo `Modal`, apresentando a tela por cima da anterior, através dos método `presentViewController` (mostrar). A responsabilidade de tirar a tela apresentada fica para quem chamou o modal, isto é, nós temos que lembrar de adicionar a `Action` para permitir fechar o modal.

Desejamos adicionar um botão “Ok” que permita fechar nossa tela. Como fizemos anteriormente, vamos instanciá-lo passando diversos parâmetros, entre os quais o título (`Ok`) e o estilo (`Cancel`):

```
let details = UIAlertController(title: meal.name,
                                message: "Happiness: \(meal.happiness)",
                                preferredStyle: UIAlertControllerStyle.Alert)
let ok = UIAlertAction(title: "Ok",
                        style: UIAlertActionStyle.Cancel,
                        /* extra parameters */)

presentViewController(details, animated: true, completion: nil)
```

Em seguida, adicionamos o botão de cancelar em nosso detalhamento:

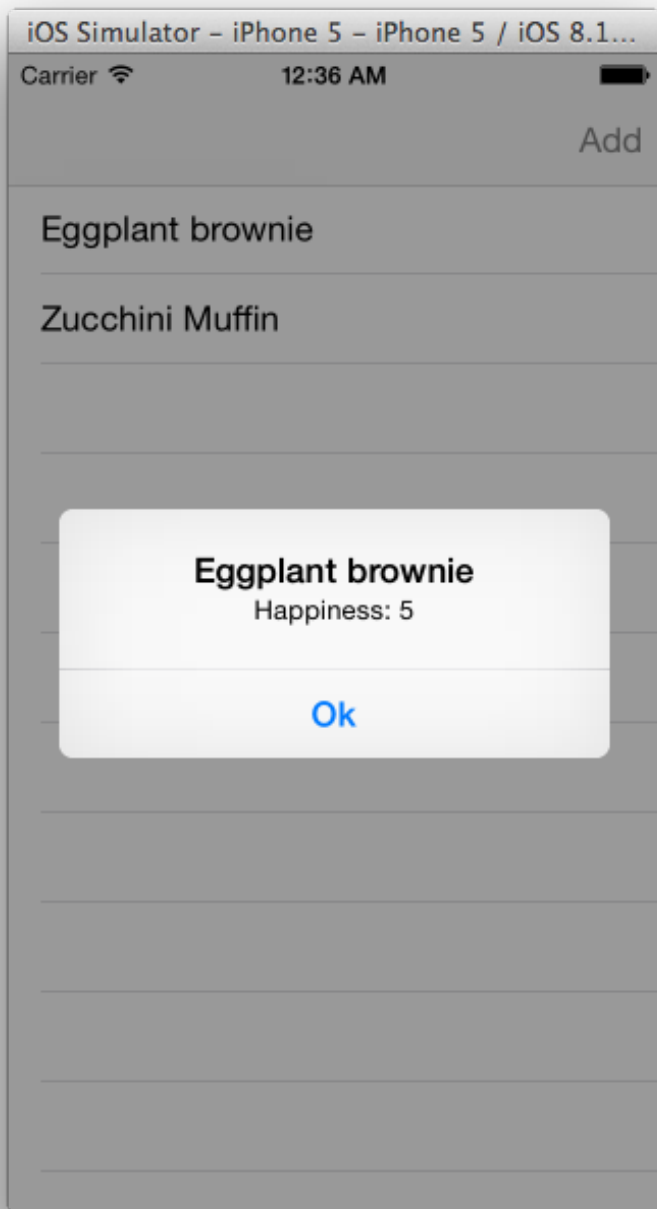
```
let details = UIAlertController(title: meal.name,
                                message: "Happiness: \(meal.happiness)",
                                preferredStyle: UIAlertControllerStyle.Alert)
```

```
let ok = UIAlertAction(title: "Ok",
    style: UIAlertActionStyle.Cancel,
    /* extra parameters */)
details.addAction(ok)
presentViewController(details, animated: true, completion: nil)
```

Mas temos ainda que falar que não queremos executar nada ao fechar nosso diálogo, passando `nil` como parâmetro:

```
let details = UIAlertController(title: meal.name,
    message: "Happiness: \(meal.happiness)",
    preferredStyle: UIAlertControllerStyle.Alert)
let ok = UIAlertAction(title: "Ok",
    style: UIAlertActionStyle.Cancel,
    handler: nil)
details.addAction(ok)
presentViewController(details, animated: true, completion: nil)
```

Rodamos e agora podemos voltar para a lista através do botão de `Ok` do nosso popup:



12.3 MOSTRANDO OS DETALHES DOS ITENS

Agora que já conseguimos mostrar os dados básicos de uma refeição, vamos acrescentar também os itens que aquela refeição tem. Pegamos todos os itens que temos na refeição e concatenamos na mensagem do popup. Como uma refeição pode ter mais do que um item, usamos um `for` para efetuarmos a concatenação. Acrescentamos no método `showDetails`:

```
var message = "Happiness: \(meal.happiness)"

for item in meal.items {
    message += "\n * \(item.name) - calories: \(item.calories)"
}

let details = UIAlertController(title: meal.name,
    message: message,
    preferredStyle: UIAlertControllerStyle.Alert)
```

Bacana, temos agora todo o detalhamento de nossas refeições mas nosso método ficou com responsabilidades demais. Além de buscar a célula selecionada, ainda precisa ser responsável por imprimir as informações e também mostrar o popup com o detalhamento. Saber as informações de detalhes é responsabilidade da própria refeição, logo, vamos extrair este comportamento para a classe `Meal`:

```
class Meal {

    // ...

    func details() -> String {
        var message = "Happiness: \(self.happiness)"

        for item in self.items {
            message
                += "\n * \(item.name) - calories: \(item.calories)"
        }

        return message
    }
}
```

Agora só precisamos utilizar este novo método dentro do `showDetails`, que ficará da seguinte forma:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

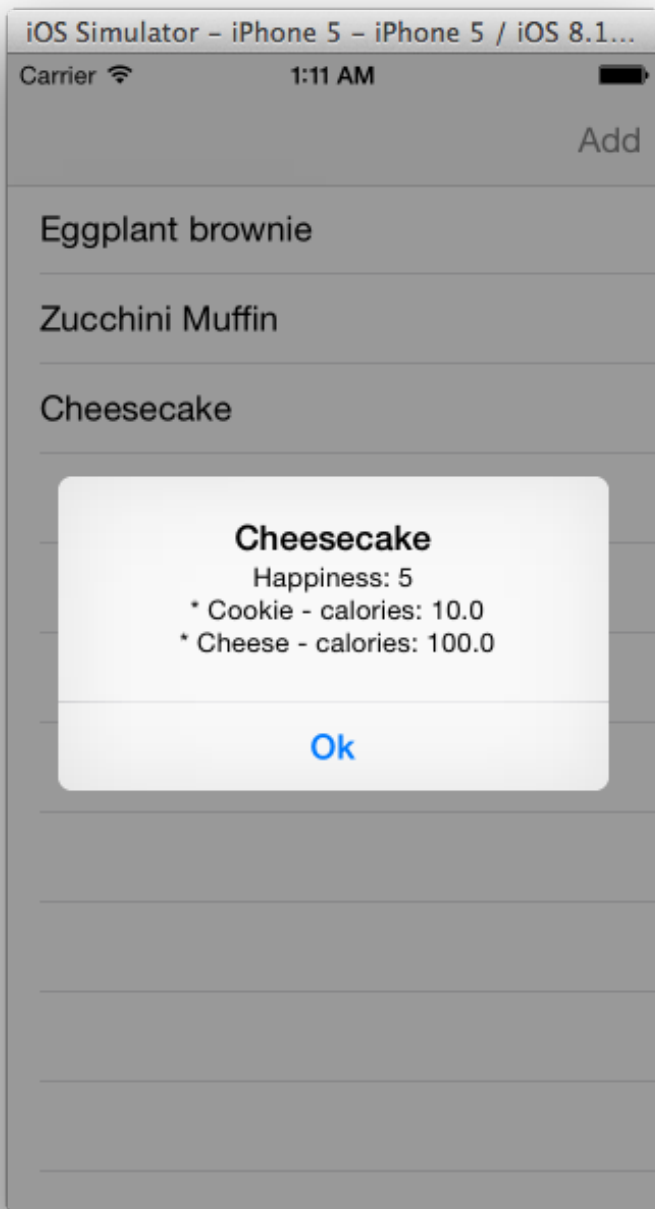
        let details = UIAlertController(title: meal.name,
                                       message: meal.details(),
                                       preferredStyle: UIAlertControllerStyle.Alert)

        let ok = UIAlertAction(title: "Ok",
                               style: UIAlertActionStyle.Cancel,
                               handler: nil)
        details.addAction(ok)

        presentViewController(
            details, animated: true, completion: nil)
    }
}
```

Nosso `controller` não precisa mais saber como fazer para imprimir as refeições. A única coisa de que ele precisa é invocar o método `details` da classe `Meal`. Esta ideia de não precisar conhecer detalhes da implementação para conseguir executar a ação desejada é um conceito muito importante da Orientação a Objetos chamado de **encapsulamento**.

Rodamos e criamos um novo item chamado `cheese` com 100 calorias. Adicionamo-lo junto do `cookie` em uma nova refeição chamada `cheesecake`. Efetuamos o `Long Press` e temos o resultado:



12.4 RESUMO

Vimos neste capítulo como podemos responder a um evento de clique longo (Long Press) e como podemos mostrar um popup personalizado utilizando um `UIAlertController`. Também vimos que podemos navegar para outras telas sem utilizar `push`, através do modo `Modal` e, no final, melhoramos nosso código, separando as responsabilidades e encapsulando as funcionalidades específicas em seus devidos lugares.

CAPÍTULO 13

Alerta: optionals e erros

Ainda temos muitos códigos espalhados que fazem verificações para detectar se as variáveis estão com valores válidos antes de explodir a aplicação.

O uso de `Optionals` em Swift é muito interessante, mas como toda referência que pode ser nula, seu uso é perigoso.

Vejamos o que podemos fazer agora para melhorar todos os `ifs` que retornam de nossas funções, e tratar os erros desconhecidos do sistema de uma maneira uniforme, de modo que o usuário entenda o que aconteceu.

Primeiro, vamos à nossa tela de nova refeição, à qual adicionamos um novo item, no `ViewController`. Repare que, ao adicionarmos um novo item em nosso array, utilizamos o código a seguir:

```
func addNew(item: Item) {  
    items.append(item)  
    if tableView == nil {
```

```

        return
    }
    tableView!.reloadData()
}

```

Qual o perigo de usar um `!`? É que sempre que usamos um `!` estamos dizendo ao compilador que sabemos o que estamos fazendo, e que a aplicação vai parar se der algum problema de acesso. O acesso a uma variável `Optional` com `!` é o equivalente ao acesso de objetos tradicionalmente feito em outras linguagens como Java. Se a referência for nula, a aplicação crasheia. Perigoso. Como não queremos isso, prometi a vocês não digitar jamais um `!`, mas acabamos usando-o aqui.

Apesar disso, como bons programadores, devemos nos proteger em todo o nosso programa. Sempre antes de usar uma referência opcional, verificamos seu valor com `if`. Qual o problema do `!`, então? É que dependemos de nós mesmos, temos que lembrar de usar o `if`, se esquecermos, ou se nos enganarmos, a aplicação crasheia.

Não queremos depender de algo tão frágil quanto nossa própria memória. Por isso, vamos evitar o `!` sempre que possível: não queremos que a aplicação crasheie e não queremos correr o risco de esquecer nosso `if`.

A solução? Só quero chamar o método `reloadData` se a variável `tableView` foi definida adequadamente, caso contrário não chame o método:

```

func addNew(item: Item) {
    items.append(item)
    tableView?.reloadData()
}

```

O próprio compilador adiciona o `if`. Só executamos o método `reloadData` se a referência para a tabela for válida. Como código, é bem menos digitação, e como bons programadores, nosso objetivo de vida é digitar o mínimo possível, certo? Errado. Não somos pagos para digitar o mínimo de caracteres, somos pagos por um produto de qualidade.

O código anterior possui um perigo tremendo: se a variável estiver definida, tudo funciona como o esperado. Se não, ele adiciona no array e não

atualiza a tabela. Inocente? Pensemos com mais cuidado: o usuário final realiza uma tarefa como efetuar uma transferência:

```
func transfer(from:Account, to:Account, value:Double) {
    from.withdraw(amount)
    to.deposit(amount)
    navigationController!.pushViewControllerAnimated(true)
}
```

A única linha que não funcionou é a que atualiza a mensagem na tela. Nessa versão, a aplicação crasheia e o usuário fica sem saber o que aconteceu. Já na versão adiante, dependemos de o desenvolvedor se lembrar de fazer um `if`:

```
func transfer(from:Account, to:Account, value:Double) {
    from.withdraw(amount)
    to.deposit(amount)
    if navigationController == nil {
        // display alert
        return
    }
    navigationController!.pushViewControllerAnimated(true)
}
```

Mudamos para o `optional chaining`, isto é, usando a `?` no lugar da `!`:

```
func transfer(from:Account, to:Account, value:Double) {
    from.withdraw(amount)
    to.deposit(amount)
    navigationController?.pushViewControllerAnimated(true)
}
```

A linha novamente não funciona, o `pop` não é feito, mas a transferência já foi efetuada. O que o usuário final faz, uma vez que não sabe que algo deu errado? Ele transfere novamente. **Caboom**. A aplicação não crasheia, mas ele efetua duas vezes uma coisa que só queria efetuar uma.

`Optional chaining` é bonito mas tão ou mais perigoso que um crash de sua aplicação. Pior ainda se existe lógica após uma chamada de `optional`

chaining! Ele potencializa o caso de esquecimento do desenvolvedor, que como qualquer ser humano está fadado ao erro.

13.1 BOA PRÁTICA: EVITE OPTIONAL, GARANTA TUDO COM IF LET

Mas se ambos são ruins, qual minha alternativa? Jamais utilizar o `!` sob qualquer circunstância e jamais usar o `?` para `optional chaining`. São palavras fortes, mas vale como quase sempre. Quase sempre vale a pena garantir que sua aplicação não terá um crash no futuro.

Só tenho variáveis obrigatórias ou opcionais definidas como `?`, que é o que forçamos até agora. Como extrair o valor de uma delas? Usando o `if let`. Se não usamos `optional chaining` nem `!`, a única maneira de extrair o valor é com o `if let`: não tem como o desenvolvedor esquecer o `if`!

```
func addNew(item: Item) {
    items.append(item)
    if let table = tableView {
        table.reloadData()
    }
}
```

Será que Swift seria uma linguagem ainda mais segura se não existissem o `optional chaining` e o `!`? Talvez, só o tempo dirá.

Claro, um desenvolvedor preguiçoso, que prefere digitar menos e correr mais risco, pode argumentar que assim estamos digitando mais. É verdade... E daí?

13.2 TRATANDO O ERRO COM UMA MENSAGEM NOVAMENTE

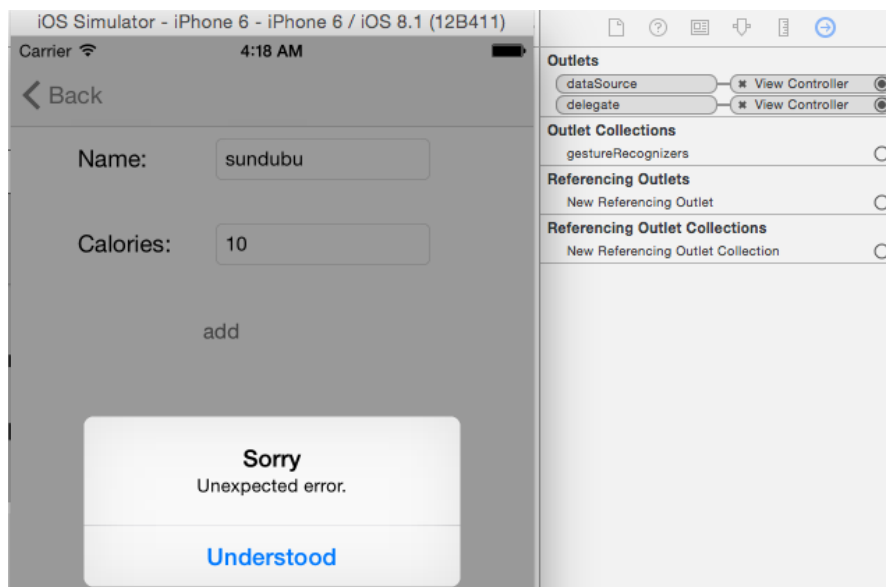
Vamos aplicar a regra que definimos dos `optionals` e tratar nossos erros. Começamos com a função que mencionamos. Quando o usuário adiciona um item, caso um erro aconteça, desejamos mostrar alguma mensagem de erro:

```
func addNew(item: Item) {
    items.append(item)
    if let table = tableView {
        table.reloadData()
    } else {
        let alert = UIAlertController(title: "Sorry",
                                      message: "Unexpected error.",
                                      preferredStyle: UIAlertControllerStyle.Alert)
        let ok = UIAlertAction(title: "Understood",
                               style: UIAlertActionStyle.Cancel,
                               handler: nil)
        alert.addAction(ok)
        presentViewController(
            alert, animated: true, completion: nil)
    }
}
```

Para testarmos, precisamos desconectar nosso *outlet*, simulando um erro de configuração ou de *runtime*. Vamos ao nosso *storyboard*, selecionamos o `UITableView` da tela de nova refeição e, na parte da direita, onde temos o `connections inspector`, removemos o `outlet` clicando no `x`.



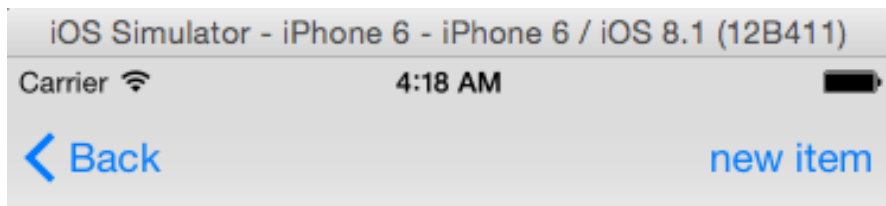
Agora testamos a aplicação e temos a mensagem de erro:



13.3 BOA PRÁTICA: MENSAGENS DE ERRO DESCRITIVAS

Apesar do erro, a mensagem pode deixar o usuário confuso. Na situação que vimos, ela indica que algo inesperado aconteceu, mas o quê? E como o usuário deve se sentir em relação a isso? Qual seu próximo passo?

Escolha suas mensagens de erro de forma bastante descritiva. Aqui, apesar de não ser possível atualizar a tabela, conseguimos adicionar o elemento em nosso array, por exemplo um `sundubu`. Ao clicarmos em `back`, vemos que o resultado havia sido positivo, mesmo tendo sido mostrada a mensagem de erro:



Name

eggplant brownie

Happiness

1-sad 5-happy

Add

Eggplant Brownie

Zucchini Muffin

Cookie

Coconut oil

Chocolate frosting

Chocolate chip

sundubu

Portanto, mudamos nossa mensagem de erro para indicar essa situação claramente, dizendo que o item foi adicionado com sucesso mas algum erro inesperado ocorreu:

```
func addNew(item: Item) {
    items.append(item)
    if let table = tableView {
        table.reloadData()
    } else {
        let alert = UIAlertController(title: "Sorry",
                                     message: "Unexpected error, but the item was added.",
                                     preferredStyle: UIAlertControllerStyle.Alert)
        let ok = UIAlertAction(title: "Understood",
                               style: UIAlertActionStyle.Cancel,
                               handler: nil)
        alert.addAction(ok)
        presentViewController(
            alert, animated: true, completion: nil)
    }
}
```

13.4 REFATORAÇÃO: TRATANDO DIVERSOS ERROS

Só de imaginar o código que colocaremos aqui já bate uma preguiça... vai ficar tão repetitivo. Não só isso, nosso controller já tem tantas responsabilidades distintas; devemos bater na tecla de uma classe, uma responsabilidade. Seria tão bom se pudéssemos resumir essa responsabilidade em uma invocação simples de mostrar mensagem de alerta:

```
func addNew(item: Item) {
    items.append(item)
    if let table = tableView {
        table.reloadData()
    } else {
        Alert().show("Unexpected error, but the item was added.")
    }
}
```

Se queremos simplificar, podemos. Extraímos o código de mostrar um *alert* de erro para uma classe específica em nosso grupo `views`:

```
class Alert {
    func show(message:String) {
        let details = UIAlertController(title: "Sorry",
                                       message: message,
                                       preferredStyle: UIAlertControllerStyle.Alert)
        let cancel = UIAlertAction(title: "Understood",
                                   style: UIAlertActionStyle.Cancel,
                                   handler: nil)
        details.addAction(cancel)
        presentViewController(
            details, animated: true, completion: nil)
    }
}
```

O código ainda não compila. Primeiro devemos importar `UIKit`, pois usamos diversos componentes de `UI` nele:

```
import UIKit
```

Agora temos um último erro de compilação, uma vez que o método `presentViewController` não existe por aqui. Ele está definido em um `UIViewController`. Precisamos de nosso controller aqui? Ótimo, recebemo-lo na inicialização, afinal, somos *good citizens*:

```
class Alert {
    let controller:UIViewController
    init(controller:UIViewController) {
        self.controller = controller
    }

    func show(message:String) {
        let details = UIAlertController(title: "Sorry",
                                       message: message,
                                       preferredStyle: UIAlertControllerStyle.Alert)
        let cancel = UIAlertAction(title: "Understood",
                                   style: UIAlertActionStyle.Cancel,
```

```
        handler: nil)
    details.addAction(cancel)
    controller.presentViewController(details,
        animated: true, completion: nil)
}
}
```

Ao criarmos nosso alerta, devemos passar o controller:

```
func addNew(item: Item) {
    items.append(item)
    if let table = tableView {
        table.reloadData()
    } else {
        Alert(controller: self).show(
            "Unexpected error, but the item was added.")
    }
}
```

Bonito. Testamos agora e, após toda essa refatoração para simplificar nosso código final, continuamos com nossa mensagem de erro funcionando. O objetivo de toda refatoração é extrair uma parte suja, simplificando-a de alguma maneira e ao mesmo tempo facilitando a utilização daquela parte por quem já a utilizava. É o que fizemos aqui: isolamos uma responsabilidade e simplificamos o código de quem precisa invocá-la.

13.5 PARÂMETROS DEFAULT

Muitas vezes utilizaremos a mesma mensagem de erro. Nesses casos, podemos utilizar um valor padrão para nosso parâmetro:

```
func show(message:String = "Unexpected error.") {
    let details = UIAlertController(title: "Sorry",
        message: message,
        preferredStyle: UIAlertControllerStyle.Alert)
    let cancel = UIAlertAction(title: "Understood",
        style: UIAlertActionStyle.Cancel,
        handler: nil)
```

```
details.addAction(cancel)
presentViewController(
    details, animated: true, completion: nil)
}
```

Agora poderíamos invocá-lo sem parâmetros:

```
func addNew(item: Item) {
    items.append(item)
    if let table = tableView {
        table.reloadData()
    } else {
        Alert(controller: self).show()
    }
}
```

Mas se desejamos manter a mensagem customizada de antes, agora que utilizamos valores `default` somos obrigados a dizer qual parâmetro estamos repassando:

```
func addNew(item: Item) {
    items.append(item)
    if let table = tableView {
        table.reloadData()
    } else {
        Alert(controller: self).show(message:
            "Unexpected error, but the item was added.")
    }
}
```

CODE SMELL: DIVERSOS PARÂMETROS COM VALOR PADRÃO

Por mais que pareça interessante utilizar diversos parâmetros com valor padrão, temos alguns cuidados a tomar. Primeiro, assim como em diversas outras linguagens, os parâmetros opcionais são sempre os últimos de uma função. Segundo, um número grande de parâmetros opcionais em geral indica uma grande quantidade de possibilidades de resultados diferentes para uma invocação à sua função e, se esse for o caso, a complexidade do método também pode estar alta.

Tome cuidado com parâmetros opcionais: se ele indica uma alta complexidade (ciclomática) do seu método, refatore. Se ele indica somente números ou `Strings` padrão que não entram em cláusulas condicionais (como `ifs`, `fors` e `switchs`), menos problemas.

Uma das maneiras de refatorar um código com muitos parâmetros opcionais (principalmente um construtor com tais características) é a utilização do *design pattern* `Builder`.

13.6 BOA PRÁTICA: SINGLE RESPONSIBILITY PRINCIPLE, PRINCÍPIO DE RESPONSABILIDADE ÚNICA

É considerada uma boa prática a utilização de uma única responsabilidade por unidade de código. Em Orientação a Objetos, é comum que os métodos estejam agrupados em uma classe de acordo com uma determinada funcionalidade. Por exemplo, os métodos de entrada e saída devem ficar em uma classe diferente que as de interface com o usuário. É por isso mesmo que criamos esse isolamento básico da classe `Alert` do resto de nosso programa.

Agora podemos aplicar nosso `Alert` a outras partes do código. Vamos primeiro ao `showNewItem` de nosso `ViewController`:

```
@IBAction func showNewItem() {
    let newItem = NewItemViewController(delegate: self)
    if let navigation = navigationController {
        navigation.pushViewController(newItem, animated:true)
    }
}
```

```
    }  
}
```

Aqui desejamos mostrar a mensagem padrão de erro:

```
@IBAction func showNewItem() {  
    let newItem = NewItemViewController(delegate: self)  
    if let navigation = navigationController {  
        navigation.pushViewController(newItem, animated:true)  
    } else {  
        Alert(controller: self).show()  
    }  
}
```

13.7 CASOS MAIS COMPLEXOS DE TRATAMENTO DE ERRO

Na mesma classe, temos a função `add`, um caso mais complicado:

```
if nameField == nil || happinessField == nil {  
    return  
}
```

E agora? Duas condições?

```
if let nameField = nameField {  
    if let happinessField = happinessField {  
        let name = nameField.text  
        let happiness = happinessField.text.toInt()  
        // ...  
    }  
}
```

IF LET X = X

A construção `if let` permite definir uma variável com o mesmo nome da variável opcional que estamos testando. Isso pode parecer uma ótima funcionalidade, mas se o nome da variável opcional é igual ao nome da variável que tem valor, isso significa que um programador desavisado pode acreditar que variável tem valor. Perigoso.

Como em Swift será necessário o uso do `!`, ou `?`, ou `let` para extrair o valor, o compilador pegará o erro do desenvolvedor (exceto em casos em que a inferência de tipo pode ser “esperta” e passar uma rasteira nele).

Em outras linguagens, é ainda mais importante que uma variável indique se seu valor é opcional ou sempre válido.

13.8 CODE SMELL: NESTED IFs

Nested ifs? Bem feito. Ainda mais agora que precisamos de um terceiro `if` para resgatar o terceiro valor, o inteiro:

```
if let nameField = nameField {
    if let happinessField = happinessField {
        let name = nameField.text
        let happiness = happinessField.text.toInt()
        if let happiness = happiness {
            // ...
        }
    }
}
```

Haja coração para aturar esse código: cinco linhas pequenas com três ifs, cinco lets, quatro names, seis happiness e seis Fields. Os nested ifs (ifs aninhados) são um indicador de que há muita responsabilidade e complexidade em nosso código. Paremos um instante e façamos a pergunta a nós mesmos: o que queremos aqui?

Dado um formulário, quero um `Meal`. Ótimo, isolemos esse comportamento, a responsabilidade de, dado um formulário `UI`, extrair um `Meal`:

```
func getMealFromForm() -> Meal {  
  
}
```

Claro, falta implementar o método, que deixaremos como estava antes:

```
func getMealFromForm() -> Meal {  
    if nameField == nil || happinessField == nil {  
        return  
    }  
  
    let name = nameField!.text  
    let happiness = happinessField!.text.toInt()  
    if happiness == nil {  
        return  
    }  
  
    let meal = Meal(name: name, happiness: happiness!)  
    meal.items = selected  
    println(  
        "eaten: \(meal.name) \(meal.happiness) \(meal.items)"  
    )  
}
```

Quando percebemos que temos algo inválido, devolvemos vazio; no caso de sucesso, devolvemos nosso `meal`:

```
func getMealFromForm() -> Meal {  
    if nameField == nil || happinessField == nil {  
        return nil  
    }  
  
    let name = nameField!.text  
    let happiness = happinessField!.text.toInt()  
    if happiness == nil {  
        return nil  
    }  
  
    let meal = Meal(name: name, happiness: happiness!)  
    meal.items = selected  
    println(  
        "eaten: \(meal.name) \(meal.happiness) \(meal.items)"  
    )  
}
```

```

        "eaten: \(meal.name) \(meal.happiness) \(meal.items)")
    return meal
}

```

Mas calma, se podemos retornar um `meal` ou não, o retorno deve ser marcado como opcional, para quem invocar esse método lembrar de tratar o caso de vazio:

```

func getMealFromForm() -> Meal? {
    if nameField == nil || happinessField == nil {
        return nil
    }

    let name = nameField!.text
    let happiness = happinessField!.text.toInt()
    if happiness == nil {
        return nil
    }

    let meal = Meal(name: name, happiness: happiness!)
    meal.items = selected
    println(
        "eaten: \(meal.name) \(meal.happiness) \(meal.items)")
    return meal
}

```

Agora invocamos o método adequadamente:

```

@IBAction func add() {
    if let meal = getMealFromForm() {
        if delegate == nil {
            return
        }

        delegate!.add(meal)

        if let navigation = self.navigationController {
            navigation.popViewControllerAnimated(true)
        }
    }
}

```

```
    }  
}
```

Podemos melhorar a validação do `delegate` com um `if let`:

```
@IBAction func add() {  
    if let meal = getMealFromForm() {  
        if let meals = delegate {  
            meals.add(meal)  
            if let navigation = self.navigationController {  
                navigation.popViewControllerAnimated(true)  
            }  
        }  
    }  
}
```

Apesar de não ser o ideal, não vamos refatorar mais nosso código. Os `nested ifs` que restaram podem ser refatorados, principalmente criando algum tipo de framework para lidar com `delegates` e navegação. O que faremos é mostrar um alerta no caso de problema de navegação:

```
@IBAction func add() {  
    if let meal = getMealFromForm() {  
        if let meals = delegate {  
            meals.add(meal)  
            if let navigation = self.navigationController {  
                navigation.popViewControllerAnimated(true)  
            } else {  
                Alert(controller: self).show(  
message: "Unexpected error, but the meal was added.")  
            }  
        }  
    }  
}
```

Em caso de sucesso, saímos da função; caso surja outra falha, mostramos a mensagem padrão de erro:

```
@IBAction func add() {  
    if let meal = getMealFromForm() {
```

```
        if let meals = delegate {
            meals.add(meal)
            if let navigation = self.navigationController {
                navigation.popViewControllerAnimated(true)
            } else {
                Alert(controller: self).show(
message: "Unexpected error, but the meal was added.")
            }
            return
        }
    }
    Alert(controller: self).show()
}
```

13.9 RESUMO

Vimos como o uso do optionals é poderoso e ao mesmo tempo extremamente perigoso. Conversamos sobre o perigo do `!` e qual o motivo para evitá-lo ao máximo. Analisamos o uso do `?` para fazer `optional chaining` e os perigos ainda mais graves em sua utilização. Por fim, optamos por usar o `if let` sempre, como a única alternativa segura ao trabalhar com valores opcionais. Vale lembrar que sempre daremos preferência para valores obrigatórios.

O resto do código criado até agora em nossos outros controllers e classes pode se beneficiar da mesma técnica, na qual evitamos *copy* e *paste* e favorecemos a extração de código comum, de responsabilidades.

Aprendemos o princípio de responsabilidade única, fundamental para facilitar a manutenção de nosso código em longo prazo.

No meio do caminho, aprendemos como criar parâmetros opcionais e os cuidados que devemos tomar com eles. Isolamos o código de visualização de alertas em uma classe que foi reutilizada em todos os pontos de nossa aplicação em que valores opcionais são encontrados.

CAPÍTULO 14

Removendo uma refeição

14.1 AÇÕES DESTRUTIVAS E ESTILOS

Nosso próximo passo é permitir que o usuário final seja capaz de remover uma refeição quando entrar com algum dado errado. Para isso, primeiro alteramos nosso botão que é referenciado através de uma variável chamada `ok` para `cancel`, além de mudar seu nome para `Cancel`:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
```

```
let meal = meals[ row ]

let details = UIAlertController(title: meal.name,
    message: meal.details(),
    preferredStyle: UIAlertControllerStyle.Alert)

let cancel = UIAlertAction(title: "Cancel",
    style: UIAlertActionStyle.Cancel,
    handler: nil)
details.addAction(cancel)

presentViewController(
    details, animated: true, completion: nil)
}
}
```

Agora adicionamos um novo botão, chamado `Remove`:

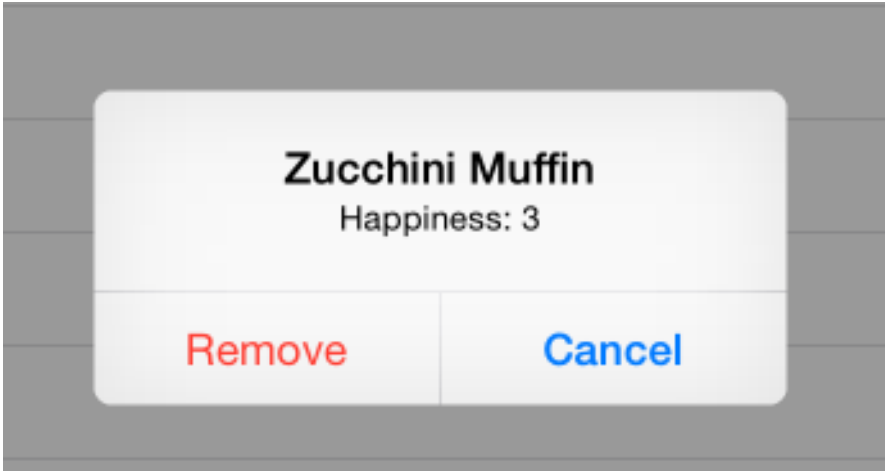
```
let remove = UIAlertAction(title: "Remove",
    style: UIAlertActionStyle.Cancel,
    handler: nil)
details.addAction(remove)
let cancel = UIAlertAction(title: "Cancel",
    style: UIAlertActionStyle.Cancel,
    handler: nil)
details.addAction(cancel)
```

Note que temos um problema: uma ação de `cancel` não é destrutiva, enquanto uma de `remove` destrói algo, é uma ação perigosa e o usuário deve entender isso. Além disso, é muito estranho que um alerta tenha duas ações de cancelar, não faz sentido! Existe um estilo chamado `Destructive` que indica que a ação será destrutiva, e é ele que usaremos.

```
let remove = UIAlertAction(title: "Remove",
    style: UIAlertActionStyle.Destructive,
    handler: nil)
details.addAction(remove)
let cancel = UIAlertAction(title: "Cancel",
    style: UIAlertActionStyle.Cancel,
```

```
handler: nil)
details.addAction(cancel)
```

Agora sim a caixa de diálogo nos diz que ao clicarmos em `Remove` algo perigoso acontecerá:



14.2 PASSANDO UMA FUNÇÃO COMO CALLBACK

Mas clicamos e nada acontece. Como assim? Claro, precisamos escrever a função que será invocada pelo iOS quando o usuário clicar em `Remove`. Vamos definir um método chamado `removeSelected` dentro de nossa classe:

```
func removeSelected() {
    println("removed the selected one")
}
```

Já passamos por outras situações em que dissemos para algum componente de `UI` que, ao efetuar uma tarefa, um `Observer` devia ser chamado, passando tanto um objeto, comumente o `self`, e o nome do método através de um `Selector` - uma `String`. `String`? Como assim? Um perigo

só. Se qualquer coisa muda, a `String` não é interpretada pelo compilador e somente descobrimos erros em tempo de execução.

Em vez de trabalharmos com `selectors`, algumas partes da API do iOS nos permitem passar um bloco de código, uma função ou um método. Podemos passar diretamente como `handler` de nossa ação uma referência para a função `removeSelected`:

```
let remove = UIAlertAction(title: "Remove",
    style: UIAlertActionStyle.Destructive,
    handler: removeSelected)
details.addAction(remove)
let cancel = UIAlertAction(title: "Cancel",
    style: UIAlertActionStyle.Cancel,
    handler: nil)
details.addAction(cancel)
```

Repare que não escrevemos `removeSelected()`, que seria o equivalente a invocar a função. Não queremos invocar a função, queremos somente passar a referência da função para o `UIAlertAction`. Por isso, passamos somente `removeSelected`.

Mas o compilador reclama. Chato. Sim, esta é uma das ideias por trás de usar um compilador: pegar em desenvolvimento problemas que quebrariam nossa aplicação em tempo de execução. Ele indica que `UIAlertAction!` não pode ser convertido para `()`. Nossa função não recebe nada, e `UIAlertAction!` não pode ser convertido para nada.

Vamos recebê-lo:

```
func removeSelected(action:UIAlertAction!) {
    println("removed the selected one")
}
```

CODE SMELL: USANDO COMPONENTES UI COMO PARÂMETROS EM OBSERVERS E O SWITCH

Qual o motivo de receber nossa própria `UIAlertAction` como parâmetro ao executar a função atrelada a uma `UIAlertAction`?

Uma função de callback como essa pode ser reutilizada por diversos eventos. Um exemplo disso é o `prepareForSegue`, um callback invocado quando qualquer `segue` de seu controller for ativado. O problema de ter uma única função para duas ações (`segues` são exemplos de ações) diferentes é que precisamos agora criar uma sequência de `ifs` que verificam valores em tempo de execução: o compilador deixa de nos ajudar e passamos a ter uma função com alta complexidade ao invés de diversas funções com pouca complexidade.

O que devemos fazer? Somente reutilize o mesmo callback, a mesma função, caso a ação a ser executada for realmente do mesmo tipo entre diversos botões, `segues` etc. Caso o código de uma ação não tenha nenhuma relação com o código de outra ação, não existe motivo para os dois estarem no mesmo método: crie duas funções e passe cada uma como argumento para quem irá invocá-la.

Inicialmente, os autores do livro acreditavam que o modelo em que o cliente que é notificado das observações (ou, hoje em dia, um *listener* que é notificado de eventos) e conhece mais sobre o objeto que está observando (modelo *push*) era um modelo que dificultava o reúso. Hoje em dia, Ralph Johnson defende em suas palestras que o modelo *observers* especializados são reutilizáveis, enquanto que genéricos não.

Sendo assim, o uso de um mesmo *observer* (ou *listener*) para diversas ações totalmente diferentes, em que recebemos como parâmetro nosso componente UI para decidir o que fazer, é um cheiro de que é possível que algo esteja ocorrendo de muito feio: muita complexidade com `ifs` e `switchs` indica um cheiro ainda maior. Evite, ajude o próximo desenvolvedor e a manutenção de seu código: cada ação distinta é uma responsabilidade diferente e merece seu próprio método.

Rodamos nossa aplicação e, agora sim, efetuamos o `long press` em um dos elemento. Temos o resultado impresso no log ao selecionarmos a opção `remove`.



14.3 IDENTIFICANDO A LINHA A SER REMOVIDA

Gostaríamos agora de imprimir o nome da refeição que será removida, somente para conferir que está tudo ok. Mas como acessar a variável que foi definida em outro método? Como o método `removeSelected` acessa a variável `meal` dentro do método `showDetails`? Complicado... Cada variável tem seu próprio escopo e não pode ser acessada fora dele:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        // code that shows the controller
    }
}

func removeSelected(action:UIAlertAction!) {
```

```

    // meal.name????????????????????????????????????????
    println("removed the selected one \(meal.name)")
}

```

Podemos criar uma variável opcional em nosso `MealsTableViewController` que representa a refeição selecionada, atribuir um valor dentro do método que seleciona e aplicá-lo no `removeSelected`. Difícil?

```

var selectedMeal:Meal?

func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]
        selectedMeal = meal

        // code that shows the controller
    }
}

func removeSelected(action:UIAlertAction!) {
    if let meal = selectedMeal {
        println("removed the selected one \(meal.name)")
    }
}

```

Complicado e feio, feio demais. Quanto mais tentamos “globalizar” nossas variáveis, e perder o escopo, perdemos o controle sobre elas. Deixá-las como opcionais? Estamos perdendo ainda mais o controle sobre o que está acontecendo e qual a situação atual de nossos objetos. Não vamos por esse caminho.

Note que a função `removeSelected` não precisa necessariamente viver dentro de nossa classe. Nossa função é um método, por ser definida

na classe, podendo ser invocada como um comportamento dos objetos do tipo `MealsTableViewController`. Mas não precisamos disso. Só precisamos dela dentro do método `showDetails`. É somente ao mostrar os detalhes de uma refeição que é necessário uma função capaz de remover refeições. O que fazer? Colocamos a função `removeSelected` dentro de nosso `showDetails`). Uma função pode existir dentro de outra, não tem problema nenhum nisso:

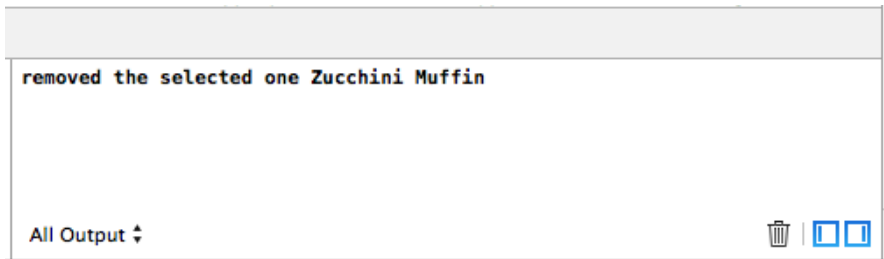
```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        func removeSelected(action:UIAlertAction!) {
            println("removed the selected one \(meal.name)")
        }

        let details = UIAlertController(title: meal.name,
            message: meal.details(),
            preferredStyle: UIAlertControllerStyle.Alert)

        let remove = UIAlertAction(title: "Remove",
            style: UIAlertActionStyle.Destructive,
            handler: removeSelected)
        details.addAction(remove)
        let cancel = UIAlertAction(title: "Cancel",
            style: UIAlertActionStyle.Cancel,
            handler: nil)
        details.addAction(cancel)
        presentViewController(
            details, animated: true, completion: nil)
    }
}
```

Removemos a definição anterior da função (não se esqueça!) e tudo continua compilando! Claro, `removeSelected` continua sendo uma referência para uma função. Antes, a função era especial, um método. Agora, a função também é especial, uma função definida dentro de nosso código. Escopo controlado, mas ambas são funções e podem ser referenciadas.



14.4 REMOVENDO E ATUALIZANDO A TELA

O próximo passo é remover de verdade usando o `removeAtIndex`:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        func removeSelected(action:UIAlertAction!) {
            println("removed the selected one \(meal.name)")
            meals.removeAtIndex(row)
        }

        // show the controller
    }
}
```

```
}
```

Testamos agora nossa aplicação, mas a tabela continua inteira. Clicamos novamente em `remove` para apagar a última refeição da lista, e a aplicação *crasheia*. O que acontece? Ele tenta acessar o array em uma posição inválida! Como assim? Já havíamos mencionado a importância de atualizarmos nossa tabela toda vez que o array for atualizado, e acabamos de cometer esse erro. Atualizamos o array mas não pedimos para a tabela ser redenhada - e ela não foi.

Mudemos nosso `removeSelected` para atualizá-la:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        // get the meal

        func removeSelected(action:UIAlertAction!) {
            println("removed the selected one \(meal.name)")
            meals.removeAtIndex(row)
            tableView.reloadData()
        }

        // show the controller
    }
}
```

Agora sim, somos capazes de remover elementos.

14.5 CLOSURES

Já podemos remover nosso `println`, que ficou desnecessário. O que mais podemos atualizar em nosso código?

Uma outra maneira de criar uma função que será utilizada poucas vezes é criá-la e já atribuí-la, seja para uma variável, seja como parâmetro na invocação de um método. Por exemplo, podemos invocar o inicializador do `UIAlertAction` já passando nosso bloco (uma *closure*) que será invocado posteriormente. Como nosso bloco recebe um `UIAlertAction` e devolve nada, marcamos como `(UIAlertAction) -> Void`:

```
let remove = UIAlertAction(title: "Remove",
    style: UIAlertActionStyle.Destructive,
    handler: {(action:UIAlertAction!) -> Void in
        meals.removeAtIndex(row)
        table.reloadData()
    })
```

Como o compilador é capaz de inferir qual o tipo do parâmetro, pois o estamos passando diretamente, não precisamos defini-lo. A mesma coisa vale para o retorno, portanto nosso código fica assim:

```
let remove = UIAlertAction(title: "Remove",
    style: UIAlertActionStyle.Destructive,
    handler: { action in
        meals.removeAtIndex(row)
        table.reloadData()
    })
```

Ótimo, mas ainda não compila. Acontece que, para uma *closure* acessar uma propriedade de nossa classe, ela precisa deixar isso explícito através do uso do `self`:

```
let remove = UIAlertAction(title: "Remove",
    style: UIAlertActionStyle.Destructive,
    handler: { action in
        self.meals.removeAtIndex(row)
        self.table.reloadData()
    })
```

Agora sim, nosso código funciona como anteriormente, e estamos usando uma *closure* em vez de uma função ou um método.

14.6 CODE SMELL: CLOSURES A RODO

Como vimos, uma função pode ser utilizada de diversas maneiras: ela pode ser declarada e utilizada diretamente como uma *closure*, pode ser definida dentro de nosso código como uma função normal ou ainda declarada em uma classe para funcionar como um método de nossos objetos. Por educação, não criamos funções globais.

Cada uma delas é aplicada com suas vantagens e desvantagens. Lembre-se: blocos e *closures* são mais difíceis de testar por compactar muito o conteúdo. São às vezes chamados de *conciso*, mas não confunda ‘conciso’ - que exige clareza, sem ambiguidades - com ‘mínimo de digitação possível’, que permite ambiguidade e por vezes dificulta a compreensão.

Por mais tentador que seja adotar o uso de *closures* em todo canto, tome **muito** cuidado. No mundo selvagem de programação você verá isso acontecendo de maneira descontrolada: usando sem dó nem piedade. Não deixe que o descontrole e o uso de diversos comportamentos em um único método o dominem: ao colocar diversas *closures* em pouco espaço, muitos comportamentos tomam conta daquele código. Fuja dessa cilada, mantenha uma responsabilidade por unidade de código.

Estamos falando tanto de muita responsabilidade e qualidade de código, mas esta classe está bem feia. Não por ser muita digitação, mas sim por ter muita responsabilidade. Note que ela é responsável por tudo ligado à view de todas as refeições, mas também à view de remover uma refeição. Onde já se viu isso? Uma classe de `view controller` que lida com dois `view controllers`. Falta de respeito por dificultar a manutenção de nosso código. Extrairemos nossa responsabilidade.

Podemos primeiro extrair um método chamado `show`, uma operação de refatoração tradicional:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        show(meal)
    }
}
func show(meal:Meal) {
```

```
let details = UIAlertController(title: meal.name,
    message: meal.details(),
    preferredStyle: UIAlertControllerStyle.Alert)

let remove = UIAlertAction(title: "Remove",
    style: UIAlertActionStyle.Destructive,
    handler: { action in
        self.meals.removeAtIndex(row)
        self.tableView.reloadData()
    })
details.addAction(remove)
let cancel = UIAlertAction(title: "Cancel",
    style: UIAlertActionStyle.Cancel,
    handler: nil)
details.addAction(cancel)
presentViewController(
    details, animated: true, completion: nil)
}
```

Mas nosso código precisa mais do que a refeição, ele precisa também do número da linha. Não vamos começar a passar diversos parâmetros picados para o método. Daqui a pouco, ele precisa de um outro `Int`, de uma `String`. Se desejamos mostrar o diálogo de remover refeição, passemos o *handler* inteiro de uma vez:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        show(meal, { action in
            self.meals.removeAtIndex(row)
            self.tableView.reloadData()
        })
    }
}
```

```

    }
}
func show(meal:Meal, handler:(UIAlertAction!) -> Void) {
    let details = UIAlertController(title: meal.name,
        message: meal.details(),
        preferredStyle: UIAlertControllerStyle.Alert)

    let remove = UIAlertAction(title: "Remove",
        style: UIAlertActionStyle.Destructive,
        handler: handler)
    details.addAction(remove)
    let cancel = UIAlertAction(title: "Cancel",
        style: UIAlertActionStyle.Cancel,
        handler: nil)
    details.addAction(cancel)
    presentViewController(
        details, animated: true, completion: nil)
}

```

Por fim, nosso método não nos pertence: vamos extraí-lo para outra classe, uma classe que representa nossa view de remover refeição. Criamos no grupo `views` o arquivo `RemoveMealController.swift`, seguindo o mesmo estilo do nosso `Alert.swift`:

```

class RemoveMealController {
    let controller:UIViewController
    init(controller:UIViewController) {
        self.controller = controller
    }
    func show(meal:Meal, handler:(UIAlertAction!) -> Void) {
        let details = UIAlertController(title: meal.name,
            message: meal.details(),
            preferredStyle: UIAlertControllerStyle.Alert)

        let remove = UIAlertAction(title: "Remove",
            style: UIAlertActionStyle.Destructive,
            handler: handler)
        details.addAction(remove)
        let cancel = UIAlertAction(title: "Cancel",

```

```
        style: UIAlertActionStyle.Cancel,
        handler: nil)
    details.addAction(cancel)
    controller.presentViewController(
        details, animated: true, completion: nil)
    }
}
```

E usamos nossa nova classe:

```
func showDetails(recognizer: UILongPressGestureRecognizer){
    if recognizer.state == UIGestureRecognizerState.Began {
        let cell = recognizer.view as UITableViewCell
        let indexPath = tableView.indexPathForCell(cell)
        if indexPath == nil {
            return
        }
        let row = indexPath!.row
        let meal = meals[ row ]

        RemoveMealController(controller: self).show(meal,
            { action in
                self.meals.removeAtIndex(row)
                self.table.reloadData()
            })
    }
}
```

Agora tudo funciona e o código está mais bem extraído. Ganhamos muito com as vantagens de refatoração e garantias de compilação de nosso código.

14.7 RESUMO

Vimos neste capítulo como permitir a remoção de uma refeição, mas, para chegar até esse ponto e mesmo após alcançá-lo, passamos por diversas melhorias de nosso código.

Aprendemos a criar um *handler* que poderia ser uma referência para uma função qualquer: desde uma *closure* até o método de um objeto específico.

Aprendemos a receber uma referência para uma função em nosso código ao extrairmos um método e uma classe para isolar melhor as responsabilidades de nosso programa. Somos capazes agora de dizer o que é uma função, e quando ela está definida como uma mera função, uma *closure* ou um método de uma classe.

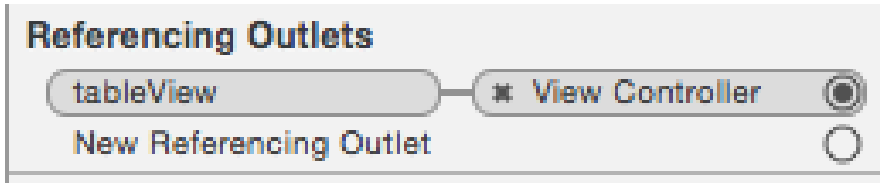
Vimos também diversos cuidados que devemos tomar e possíveis refatorações a serem efetuadas para melhorar nosso código a cada novo passo que damos.

CAPÍTULO 15

Armazenando as refeições dentro do file system

Conseguimos criar e mostrar a lista de refeições, mas o que acontece quando fechamos a aplicação? Ao rodarmos novamente, vemos que os dados que estavam inseridos não aparecem mais! Queremos que, uma vez que as refeições e os itens sejam cadastrados, eles continuem dentro da app. Para isso, precisamos deixar de salvar nossos dados em memória e salvarmos no sistema de arquivos do aparelho toda vez que sairmos de nossa aplicação.

Primeiro, vamos reconectar nossa tabela de itens para que ela volte a funcionar. Para isso, voltamos a selecionar a nossa `TableView` e conectamos seu `IBOutlet` com nossa variável no controller.



Uma classe que nos possibilita fazer isso é a `NSKeyedArchiver`. Ela permite converter qualquer objeto para um formato que pode ser gravado em um arquivo no sistema de arquivos, temos apenas que dizer quais dados de nosso objeto queremos que sejam armazenados. Mas como o `NSKeyedArchiver` sabe qual objeto pode e qual não pode ser gravado?

Para garantir que conseguimos responder a este formato, temos que implementar um método específico com o qual o `NSKeyedArchiver` sabe trabalhar. É o que fazemos utilizando o protocolo `NSCoding` em nosso objeto `Meal`:

```
class Meal : NSCoding {
    // ...
}
```

Para o protocolo funcionar, precisamos que nossa classe herde os comportamentos da classe `NSObject`. Portanto, na classe `Meal` vamos herdar de `NSObject` e assinar o protocolo `NSCoding`:

```
class Meal : NSObject, NSCoding {
    // ...
}
```

O compilador irá reclamar que não implementamos o método do protocolo, apesar de dizermos que o adotamos. Claro! Já sabemos como protocolos funcionam. Mas quais os comportamentos que precisamos ter? Primeiro, o `NSKeyedArchiver` deve ser capaz de transformar nossos objetos em algum valor que possa ser salvo, portanto precisamos de um método que, ao ser executado em um objeto, salva os dados dele, um método que `encode` nosso

objeto, que o serialize. Depois, precisaremos de um comportamento que permita desserializar, recuperar o objeto para a memória dadas as informações que estavam (em nosso caso) em um arquivo. Isso é, precisamos inicializar um objeto e ler os dados que já existiam, em outras palavras precisaremos de um inicializador para a deserialização.

Vamos colocar o método `encodeWithCoder`, que é o responsável por transformar nosso objeto `Meal` em algo que possa ser gravado em disco, ou seja, encodar o nosso objeto. Esse método recebe um objeto do tipo `NSCoder` onde jogamos os dados de nosso objeto passando uma chave e o valor correspondente:

```
class Meal : NSObject, NSCoder {
    // ...

    func encodeWithCoder(aCoder: NSCoder) {
        aCoder.encodeObject(self.name, forKey: "name")
        aCoder.encodeInteger(self.happiness, forKey: "happiness")
        aCoder.encodeObject(self.items, forKey: "items")
    }
}
```

Sabemos como encodar, mas e como buscaremos os dados do arquivo? Temos que fazer a volta! O protocolo `NSCoding` também define um inicializador para isso, para criar um objeto utilizando os dados do disco. Colocamos o inicializador na nossa classe `Meal` decodando os dados a partir do `NSCoder`:

```
class Meal : NSObject, NSCoder {
    // ...

    init(coder aDecoder: NSCoder) {
        self.name = aDecoder.decodeObjectForKey("name") as String
        self.happiness =
            aDecoder.decodeIntegerForKey("happiness")
        self.items =
            aDecoder.decodeObjectForKey("items") as Array<Item>
    }
}
```

```
func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(self.name, forKey: "name")
    aCoder.encodeInteger(self.happiness, forKey: "happiness")
    aCoder.encodeObject(self.items, forKey: "items")
}
}
```

Deixando dessa forma, o Xcode vai reclamar pois o protocolo obriga que o construtor seja implementado por todas as classes, logo, precisamos colocar a palavra `required` na assinatura do método:

```
class Meal : NSObject, NSCoding {
    // ...

    required init(coder aDecoder: NSCoder) {
        self.name = aDecoder.decodeObjectForKey("name") as String
        self.happiness =
            aDecoder.decodeIntegerForKey("happiness")
        self.items =
            aDecoder.decodeObjectForKey("items") as Array<Item>
    }

    func encodeWithCoder(aCoder: NSCoder) {
        aCoder.encodeObject(self.name, forKey: "name")
        aCoder.encodeInteger(self.happiness, forKey: "happiness")
        aCoder.encodeObject(self.items, forKey: "items")
    }
}
```

BOA PRÁTICA: DESSERIALIZAÇÃO NA INICIALIZAÇÃO

De qual outra maneira poderíamos ter feito, em vez de implementar a desserialização no construtor? Duas opções são bem comuns: uma em que implementamos um método, algo como `decodeWithDecoder`:

```
func decodeWithDecoder(aDecoder: NSCoder) {
    self.name = aDecoder.decodeObjectForKey("name") as String
    self.happiness =
        aDecoder.decodeIntegerForKey("happiness")
    self.items =
        aDecoder.decodeObjectForKey("items") as Array<Item>
}
```

Mas note que nessa abordagem seria impossível criarmos `good citizens`. A outra é a criação de um segundo objeto capaz de serializar e desserializar dados, alguém responsável pelo processo de serialização e desserialização de um modelo (algo como uma *factory* e *de-factory*, um *converter*):

```
class MealConverter {
    func decodeWithDecoder(aDecoder: NSCoder) -> Meal {
        let name = aDecoder.decodeObjectForKey("name") as String
        let happiness = aDecoder.decodeIntegerForKey("happiness")
        let items =
            aDecoder.decodeObjectForKey("items") as Array<Item>
        let meal = Meal(name, happiness)
        meal.items = items
        return meal
    }

    func encodeWithCoder(meal:Meal, aCoder: NSCoder) {
        aCoder.encodeObject(meal.name, forKey: "name")
        aCoder.encodeInteger(meal.happiness, forKey: "happiness")
        aCoder.encodeObject(meal.items, forKey: "items")
    }
}
```

Enquanto na desserialização ainda temos um *good citizen*, a abordagem do `encodeWithCoder` quebra a regra básica de encapsulamento,²³⁵ ao obrigar a conhecer tudo que uma refeição possui ao converter.

É-book gerado especialmente para Gelson Gomes gelson_agf@hotmail.com
 Levando em conta o *trade-off* entre quebrar o *good citizen* e quebrar de encapsulamento, a abordagem do `NSCoding` é a de

GOOD CITIZEN E O ARRAY DE ITENS

Até agora usamos a definição dos itens de uma refeição após a criação de um `Meal`. Continuaremos assim pelo nosso projeto, mas é claramente visível que, se uma refeição não tivesse seus itens alterados nunca, os mesmos poderiam ser configurados na inicialização, podendo manter a definição da variável `items` com um `let`, uma constante.

INICIALIZANDO UM VIEW CONTROLLER COM NSCoder

Agora podemos entender que quando herdamos de um `UIViewController` ganhamos de graça a capacidade de serializar nossos `controllers` e é justamente por isso que, ao criarmos um novo inicializador, devemos garantir que a inicialização também ocorrerá educadamente caso o `controller` seja deserializado, isto é, o `init(coder aDecoder: NSCoder)` seja invocado.

Podemos fazer um `Command+Click` no nome do protocolo, `NSCoding` e revisar como ele foi definido. Repare que ele possui as duas características mencionadas anteriormente: tanto o método `encodeWithCoder` quanto o `init` devem ser definidos.

Ensinaamos como fazemos para encodar uma refeição, mas a refeição é composta por itens, logo, devemos ensinar como encodar os itens também. Na classe `Item` repetimos o mesmo processo que fizemos na classe `Meal`. Ela ficará da seguinte forma:

```
class Item: NSObject, Equatable, NSCodering {
    let name:String
    let calories:Double
    init(name: String, calories: Double) {
        self.name = name
        self.calories = calories
    }
}
```

```
required init(coder aDecoder: NSCoder) {
    self.name = aDecoder.decodeObjectForKey("name") as String
    self.calories = aDecoder.decodeDoubleForKey("calories")
}

func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(self.name, forKey: "name")
    aCoder.encodeDouble(self.calories, forKey: "calories")
}
}
```

15.1 SALVANDO AS REFEIÇÕES NO SISTEMA DE ARQUIVOS

Agora que conseguimos transformar os objetos, chegou o momento de salvarmos os dados no nosso `file system`. Toda vez que criamos uma nova refeição, queremos salvar os dados de nossas refeições. Alteramos o método `add` de nosso `MealsTableViewController` para salvar os dados no arquivo usando o `NSKeyedArchiver` e tiramos as refeições que havíamos colocado anteriormente, inicializando com um `Array` vazio:

```
var meals = Array<Meal>()

func add(meal: Meal) {
    meals.append(meal)
    NSKeyedArchiver.archiveRootObject(meals, toFile: archive)
    tableView.reloadData()
}
```

Precisamos saber onde criar o arquivo que queremos gravar e cujos dados queremos ler. Só podemos efetuar a leitura e a escrita de arquivos que estejam dentro do diretório de nossa aplicação, portanto obtemos o caminho para o diretório através da função `NSSearchPathForDirectoriesInDomains`.

```
override func viewDidLoad() {
    let userDirs = NSSearchPathForDirectoriesInDomains(
        ?,
```

```

        ?,
        ?)
    }

```

Mas a função recebe três parâmetros. O primeiro indica que tipo de diretório estamos procurando, o diretório de documentos. O segundo indica qual domínio estamos procurando, o de usuários. Por fim, o terceiro argumento diz se desejamos que o caminho para o diretório seja absoluto ou relativo à *home* do usuário. Desejamos absoluto:

```

override func viewDidLoad() {
    let userDirs = NSSearchPathForDirectoriesInDomains(
        NSSearchPathDirectory.DocumentDirectory,
        NSSearchPathDomainMask.UserDomainMask,
        true)
}

```

Invocar `NSSearchPathForDirectoriesInDomains` retorna, um array com todos os diretórios de usuários, mas como no iOS temos somente um usuário, podemos pegar sempre a primeira posição deste array. Montamos o nome do arquivo e, como ele será necessário tanto para ler quanto para escrever, criamos o diretório uma única vez dentro do método `viewDidLoad`.

```

override func viewDidLoad() {
    let userDirs = NSSearchPathForDirectoriesInDomains(
        NSSearchPathDirectory.DocumentDirectory,
        NSSearchPathDomainMask.UserDomainMask,
        true)
    let dir = userDirs[ 0 ] as String
    let archive = "\(dir)/eggplant-brownie-meals"
}

```

Já podemos ler as refeições do arquivo caso ele já exista:

```

var meals = Array<Meal>()

override func viewDidLoad() {
    let userDir = NSSearchPathForDirectoriesInDomains(
        NSSearchPathDirectory.DocumentDirectory,

```

```
        NSSearchPathDomainMask.UserDomainMask,
        true)
    let dir = userDir[ 0 ] as String
    let archive = "\(dir)/eggplant-brownie-meals"
    if let loaded =
        NSKeyedUnarchiver.unarchiveObjectWithFile(archive) {
        self.meals = loaded as Array
    }
}
```

E no nosso método de salvar, usamos o mesmo arquivo:

```
func add(meal: Meal) {
    meals.append(meal)
    let userDir = NSSearchPathForDirectoriesInDomains(
        NSSearchPathDirectory.DocumentDirectory,
        NSSearchPathDomainMask.UserDomainMask,
        true)
    let dir = userDir[ 0 ] as String
    let archive = "\(dir)/eggplant-brownie-meals"
    NSKeyedArchiver.archiveRootObject(meals, toFile: archive)
    tableView.reloadData()
}
```

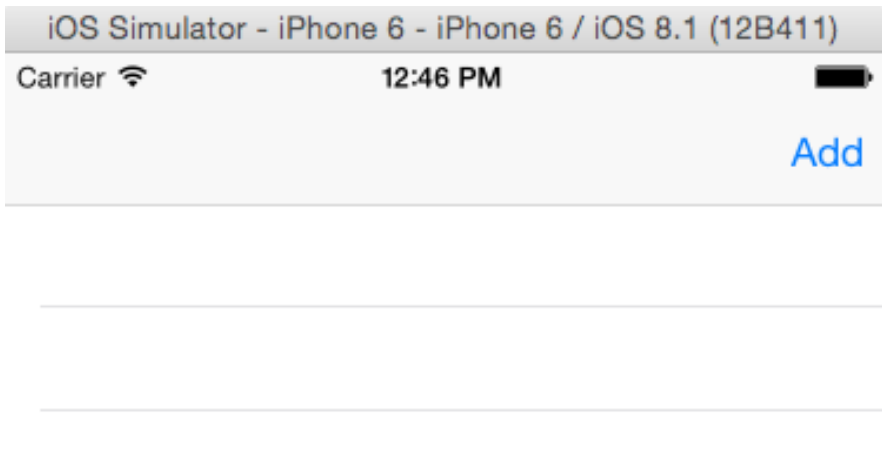
Mas... *Copy e paste* mesmo? Vamos refatorar nosso código e extrair um método, o `getUserDir`:

```
func add(meal: Meal) {
    meals.append(meal)
    let dir = getUserDir()
    let archive = "\(dir)/eggplant-brownie-meals"
    NSKeyedArchiver.archiveRootObject(meals, toFile: archive)
    tableView.reloadData()
}

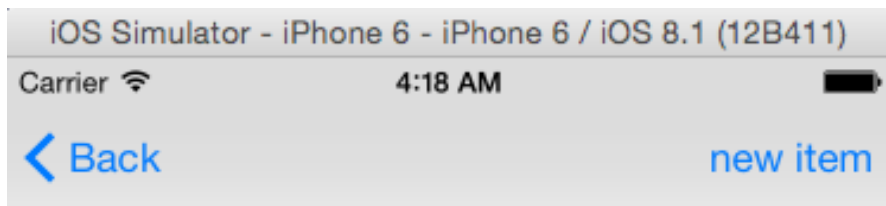
func getUserDir() -> String {
    let userDir = NSSearchPathForDirectoriesInDomains(
        NSSearchPathDirectory.DocumentDirectory,
        NSSearchPathDomainMask.UserDomainMask,
        true)
    return userDir[ 0 ] as String
}
```

```
}  
override func viewDidLoad() {  
    let dir = getUserDir()  
    let archive = "\(dir)/eggplant-brownie-meals"  
    if let loaded =  
        NSKeyedUnarchiver.unarchiveObjectWithFile(archive) {  
        self.meals = loaded as Array  
    }  
}
```

Agora vamos rodar nossa aplicação. Vemos a tabela limpa:



Adicionamos uma nova refeição: um *Sundubu* (tofu coreano). No menu de Hardware, escolhemos a Home (ou restartamos o simulador do zero) e podemos ver que nossa refeição ainda está armazenada.



Name

Happiness

Add

Eggplant Brownie

Zucchini Muffin

Cookie

Coconut oil

Chocolate frosting

Chocolate chip

sundubu

15.2 SALVANDO E LENDO ITENS NO SISTEMA DE ARQUIVOS

Conseguimos gravar as refeições, mas precisamos gravar também os itens separadamente. Vamos efetuar a mesma alteração em nosso `ViewController`. Primeiro, inicializamos o `Array` de itens vazio:

```
var items = Array<Item>()
```

Obtemos o caminho para o arquivo e montamos o nome dele dentro do `viewDidLoad`:

```
func getUserDir() -> String {
    let userDir = NSSearchPathForDirectoriesInDomains(
        NSSearchPathDirectory.DocumentDirectory,
        NSSearchPathDomainMask.UserDomainMask,
        true)
    return userDir[ 0 ] as String
}

override func viewDidLoad() {
    // Creating button...

    let dir = getUserDir()
    let archive = "\(dir)/eggplant-brownie-items"
}
```

Carregamos também os dados do arquivo dentro de nossa variável `items`:

```
override func viewDidLoad() {
    // Creating button...

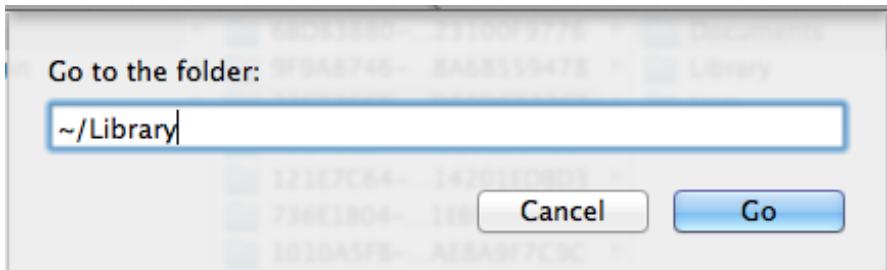
    let dir = getUserDir()
    let archive = "\(dir)/eggplant-brownie-items"

    if let loaded =
        NSKeyedUnarchiver.unarchiveObjectWithFile(archive) {
        items = loaded as Array
    }
}
```

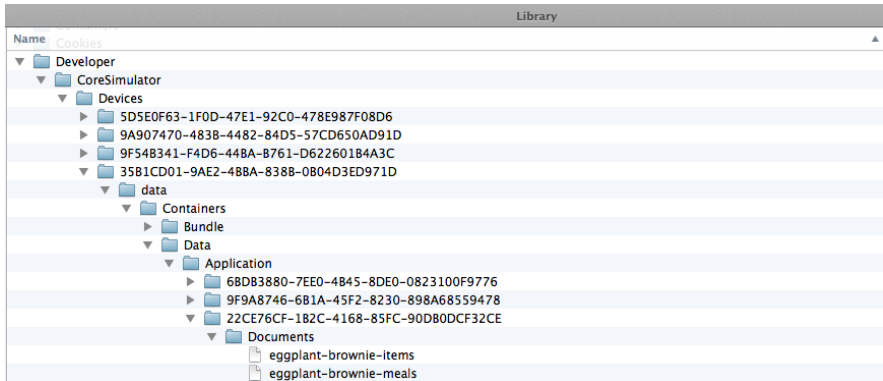
Por fim, salvamos os dados no arquivo dentro do método `addNew`:

```
func addNew(item: Item) {
    items.append(item)
    let dir = getUserDir()
    let archive = "\(dir)/eggplant-brownie-items"
    NSKeyedArchiver.archiveRootObject(items, toFile: archive)
    if let table = tableView {
        table.reloadData()
    } else {
        Alert(controller: self).show(message:
            "Unexpected error, but the item was added.")
    }
}
```

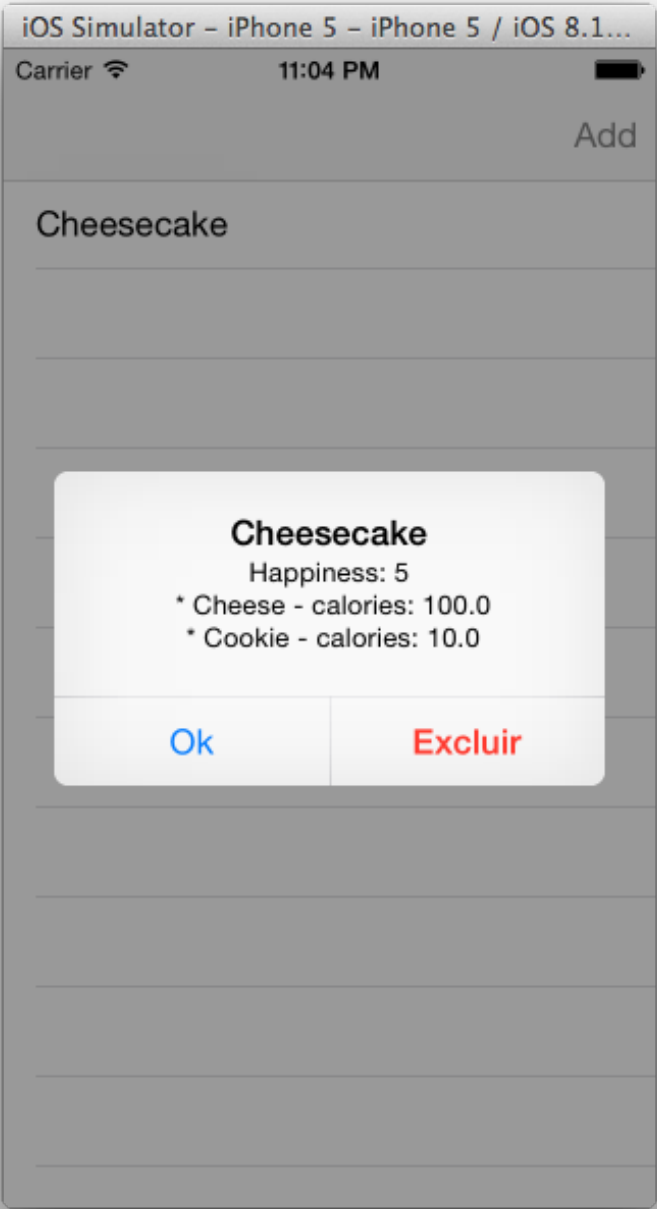
Conseguimos acessar o diretório `Library` para procurarmos onde o arquivo foi gerado, fazendo `cmd+shift+g` no `Finder` e digitando `~/Library`:



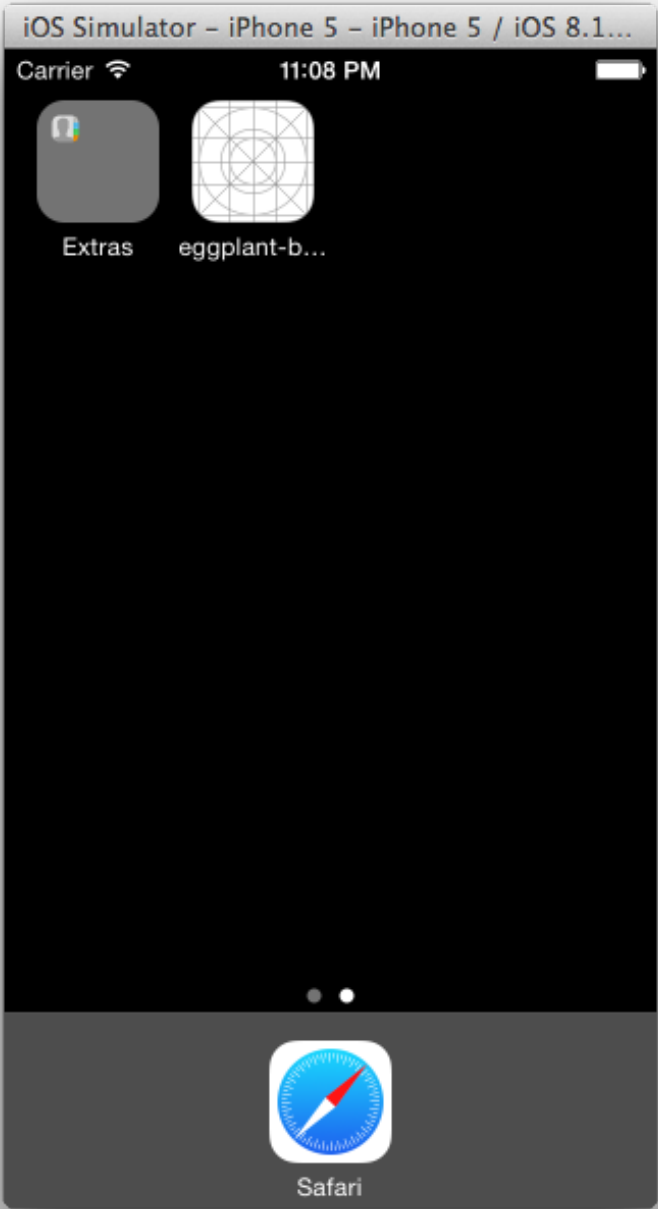
O arquivo será gerado em um caminho parecido com o seguinte:



Como agora os dados pré-cadastrados não existem mais, rodamos e criamos novamente um novo item chamado *cheese* com 100 calorias, um item chamado *cookie*, e adicionamos os dois em uma nova refeição chamada *cheesecake* com nível de felicidade 5. Ao efetuarmos o `long press`, podemos ver que os dados foram cadastrados com sucesso:



Para verificarmos se os itens continuam lá mesmo após sairmos de nossa aplicação, é só fazer `cmd+shift+h` no simulador, que ele fechará a aplicação:



Ao clicarmos na aplicação, podemos ver que a refeição adicionada continua lá.

Resumo

Neste capítulo, vimos como transformar objetos para serem salvos e recuperados através do protocolo `NSCoding` e também como manipular arquivos utilizando `NSKeyedArchiver` e `NSKeyedUnarchiver`.

Aprendemos como o protocolo `NSCoding` obriga não só a implementação de um método como também a existência de um inicializador do tipo `required`. Vimos também como, ao sair e voltar a nossa aplicação, os dados foram carregados do sistema de arquivos, inclusive sendo capazes de localizá-los em nosso computador.

No trajeto até aqui extraímos uma função que nos auxilia a definir o diretório de armazenamento de dados, mas ainda parece que temos muito `copy` e `paste`, algo que devemos atacar no próximo capítulo.

CAPÍTULO 16

Boa prática: dividindo responsabilidades e o Data Access Object

Conseguimos salvar e buscar os dados com sucesso, porém nosso controller acabou ficando com muitas responsabilidades: além de representar e responder as ações da view e de interagir com outras, agora ele é responsável por armazenar e ler os dados do disco. Temos que pensar ainda que cada controller está cuidando de seus próprios dados, repetindo, por exemplo, a lógica para buscarmos o diretório onde criamos os nossos arquivos. E se algum dia resolvermos armazenar os dados de uma outra forma? Teremos que alterar em todos os controllers esta nossa lógica. Para não termos este problema, vamos criar uma outra classe no grupo `models`, que terá a responsabilidade

de trabalhar com o armazenamento de dados. Para isso, escolhemos o menu File, New, iOS, Source, Swift File e damos o nome de Dao:

```
class Dao {  
  
}
```

Em nosso Dao, criamos as variáveis que vão receber o nome de nossos arquivos:

```
class Dao {  
    let mealsArchive: String  
    let itemsArchive: String  
}
```

Vamos efetuar também o carregamento dos dados no momento em que inicializarmos o Dao. Criamos o construtor para ele, onde buscamos o diretório do usuário uma única vez e montamos os nomes dos dois arquivos que utilizaremos:

```
init(){  
    let userDir = NSSearchPathForDirectoriesInDomains(  
        NSSearchPathDirectory.DocumentDirectory,  
        NSSearchPathDomainMask.UserDomainMask,  
        true)  
    let dir = userDir[0] as String  
    mealsArchive = "\(dir)/eggplant-brownie-meals"  
    itemsArchive = "\(dir)/eggplant-brownie-items"  
}
```

Agora criamos o método para salvar e carregar as refeições:

```
func saveMeals(meals: Array<Meal>){  
    NSKeyedArchiver.archiveRootObject(  
        meals, toFile: mealsArchive)  
}  
func loadMeals() -> Array<Meal> {  
    if let loaded =  
        NSKeyedUnarchiver.unarchiveObjectWithFile(mealsArchive) {  
        return loaded as Array
```

```
    }  
}
```

Mas precisamos cuidar do caso de o arquivo não existir, em que retornamos um array vazio:

```
func loadMeals() -> Array<Meal> {  
    if let loaded =  
        NSKeyedUnarchiver.unarchiveObjectWithFile(mealsArchive) {  
        return loaded as Array  
    }  
    return Array<Meal>()  
}
```

Vamos salvar e carregar itens:

```
func saveItems(items: Array<Item>){  
    NSKeyedArchiver.archiveRootObject(  
        items, toFile: itemsArchive)  
}  
func loadItems() -> Array<Item> {  
    if let loaded =  
        NSKeyedUnarchiver.unarchiveObjectWithFile(itemsArchive) {  
        return loaded as Array  
    }  
    return Array<Item>()  
}
```

UM DAO POR MODELO

Em aplicações maiores, é comum utilizar uma classe de DAO por modelo, ou ainda, algum outro tipo de divisão de responsabilidades de salvar e carregar dados de uma fonte. No nosso projeto, em que possuímos somente quatro métodos de uma linha, não há a necessidade de refinar ainda mais a responsabilidade de tais classes. Como um bom programador, sempre julgue com cuidado o momento no qual acredita ser adequada a quebra de responsabilidades.

Code smell: nome do tipo no nome da variável, método etc

Repare qual a palavra que se repete em cada uma das linhas; e quantas vezes?

```
func saveItems(items: Array<Item>)  
func saveMeals(meals: Array<Meal>)  
func loadMeals() -> Array<Meal>  
func loadItems() -> Array<Item>
```

Em uma linguagem como Swift, o próprio compilador é capaz de dizer qual método está sendo invocado de acordo com os tipos dos argumentos passados. Portanto, não há necessidade de repetir o nome do tipo recebido como parâmetro no nome de um método. As funções de `save` poderiam ser:

```
func save(items: Array<Item>)  
func save(meals: Array<Meal>)  
func loadMeals() -> Array<Meal>  
func loadItems() -> Array<Item>
```

Mas imagine como ficaria estranho se o mesmo ocorresse com o método `load`, que dependeria do tipo da variável ao qual o retorno está sendo aplicado, para o compilador entender qual método está sendo invocado. Complexo. Por esse motivo, manteremos os padrões `loadMeals` e `loadItems`. Já que mantemos esse padrão no `load`, manteremos o mesmo no `save`.

O dia em que você como desenvolvedor julgar adequado quebrar o código em dois `Daos`, poderá tranquilamente renomear os métodos para evitar repetição de nome de tipo em nome de método.

Agora que o `Dao` está pronto, podemos alterar nossos controllers para utilizar este novo objeto. Primeiro, mudamos a classe `ViewController` para apenas chamar o método `loadItems` no `viewDidLoad`, removendo o código anterior onde buscávamos o diretório:

```
override func viewDidLoad() {  
    let newItemButton = UIBarButtonItem(title: "new item",  
                                        style: UIBarButtonItemStyle.Plain,  
                                        target: self,  
                                        action: Selector("showNewItem"))
```

```
        navigationItem.rightBarButtonItem = newItemButton
        items = Dao().loadItems()
    }
```

No momento em que salvamos os dados, vamos chamar o método `saveItems` para guardar os itens no arquivo:

```
func addNew(item: Item) {
    items.append(item)
    Dao().saveItems(items)
    if let table = tableView {
        table.reloadData()
    } else {
        Alert(controller: self).show(message:
            "Unexpected error, but the item was added.")
    }
}
```

Com o `ViewController` certo, vamos arrumar agora o `MealsTableViewController` para buscar e salvar as refeições utilizando os métodos `loadMeals` e `saveMeals`, removendo todo o código anterior de manipulação do arquivo:

```
override func viewDidLoad() {
    meals = Dao().loadMeals()
}

func add(meal: Meal) {
    meals.append(meal)
    Dao().saveMeals(meals)
    tableView.reloadData()
}
```

Rodamos novamente e nossa aplicação continua funcionando da mesma forma.

Resumo

Tiramos responsabilidades exageradas de nossos controllers, isolando a funcionalidade de manipulação do arquivo, que na verdade é a de acesso aos

dados, em uma única classe. Ainda não os deixamos com uma única responsabilidade, mas essa refatoração é mais um passo a caminho de um código mais fácil de se manter. Aprendemos com isso o padrão de projeto chamado `Data Access Object`, o DAO.

Vimos também quando devemos quebrar essa responsabilidade em pedaços ainda menores e como a nomenclatura de métodos e variáveis pode influenciar a legibilidade de nosso código.

CAPÍTULO 17

Aonde chegamos e próximos passos

Passamos juntos por uma longa jornada. Juntos pois o livro foi escrito à medida que a linguagem se desenvolvia, desde suas versões beta (junto com o Xcode e Yosemite também beta), com muitas mudanças no caminho.

A visão geral da linguagem e mais aprofundada de alguns tópicos cria aqui uma base da linguagem que nos permite primeiramente escrever um código mais bonito. Não estou falando de beleza por ter menos linhas ou ser funcional. A beleza de ter um código que funciona e que foi escrito com outros desenvolvedores em mente - com a preocupação da manutenção.

A utilização de boas práticas e padrões de projeto nos ajudam a atingir tais objetivos.

A partir de agora, descobrir novas funcionalidades da linguagem, explo-

rar novas APIs, utilizar novas bibliotecas etc. passa a ser seu trabalho do dia a dia. Coloque tudo o que viu até aqui em prática, e que a nova geração de desenvolvedores iOS aproveite as melhores características da linguagem e da API para a mudança do mundo em que vivemos.

Boa jornada.

Índice Remissivo

- Boa prática ou Code smell? O Storyboard é uma solução do bem ou do mal?, [154](#)
- Boa prática: constantes, [42](#)
- Boa prática: crie um diretório por grupo, [74](#)
- Boa prática: cuidado com inferência de tipos, [52](#)
- Boa prática: desserialização na inicialização, [234](#)
- Boa prática: encapsulamento, [69](#)
- Boa prática: evite optional, garanta tudo com if let, [200](#)
- Boa prática: Good citizen (evite nulos), [63](#)
- Boa prática: mensagens de erro descritivas, [202](#)
- Boa prática: organizando os dados, [55](#)
- Boa prática: Orientação a Objetos, [56](#)
- Boa prática: placeholder e keyboard type, [23](#)
- Boa prática: recarregue os dados de uma tabela, [131](#)
- Boa prática: Single Responsibility Principle, Princípio de Responsabilidade Única, [208](#)
- Code smell (mal cheiro de código):
opcional, [70](#)
- Code smell: cast, [183](#)
- Code smell: closures a rodo, [225](#)
- Code smell: comentários longos, [43](#)
- Code smell: Como não aprender Orientação a Objetos: Getters e Setters, [64](#)
- Code smell: Nested ifs, [210](#)
- Code smell: nomes genéricos demais e como extrair um protocolo, [131](#)
- Code smell: usando componentes UI como parâmetros em observers e o switch, [218](#)
- Design pattern: delegate, [123](#)
- Má prática: muitos segues com parâmetros, [135](#)